The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture

Charles Richard Lang, Jr

California Institute of Technology
Computer Science Department

5014:TR:82

The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture

Thesis by

Charles Richard Lang, Jr.

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

Computer Science Department

Technical Report Number 5014

California Institute of Technology

Pasadena, California

1982

(Submitted May 24, 1982)

Acknowledgements

This thesis owes its completion to my adviser, Chuck Seitz, who both pushed me along in the right direction and provided needed encouragement and criticism of my ideas. I owe thanks to Mike Ullner and Jim Kajiya for their advice concerning programming languages and for their friendship. I also appreciate the friendship of Dan Whelan who had to listen to a lot of my griping along the way. I would like to thank Alain Martin for his many good suggestions and incisive critique of this thesis.

For financial support I thank DARPA which sponsored this research and most of my work at Caltech.

The work described in this document was sponsored by the Defense Advanced Research Projects Agency, ARPA order 3771, and monitored by the Office of Naval Research under contract N00014-79-C-0597.

Abstract

A homogeneous machine architecture, consisting of a regular interconnection of many identical elements, exploits the economic benefits of VLSI technology. A concurrent programming model is presented that is related to object oriented languages such as Simula and Smalltalk. Techniques are developed which permit the execution of general purpose object oriented programs on a homogeneous machine. Both the hardware architecture and the supporting software algorithms are demonstrated to scale their performance with the size of the system.

The program objects communicate by passing messages. Objects may move about in the system and may have an arbitrary pointer topology. A distributed, on-the-fly garbage collection algorithm is presented which operates by message passing. Simulation of the algorithm demonstrates its ability to collect obsolete objects over the entire machine with acceptable overhead costs. Algorithms for maintaining the locality of object references and for implementing a virtual object capability are also presented.

To insure the absence of hardware bottlenecks, a number of interconnection strategies are discussed and simulated for use in a homogeneous machine. Of those considered, the Boolean N-cube connection is demonstrated to provide the necessary characteristics.

The object oriented machine will provide increased performance as its size is increased. It can execute a general purpose, concurrent, object oriented language where the size of the machine and its interconnection topology are transparent to the programmer.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Homogeneous Machines

This thesis addresses the design and use of a class of ensemble architectures [Seitz82] called homogeneous machines. A homogeneous machine is a collection of nominally identical processors, each with its own storage, executing programs concurrently, and passing messages over a regular communication structure.

This basic hardware model, shown in Figure 1-1, makes no presumption about the topology or bandwidth of the communication structure, nor about details of the processors such as their instruction sets. However, the performance of such a system will depend directly on the performance of the individual processors and of the communication structure. The case of a single computer or collection of personal computers on a local network would fit this hardware model, and programs written for a large ensemble should certainly be executable on a single machine or collection of personal computers. However, the hardware environment under consideration here is one containing a large number of nominally identical processors, typically thousands, or from as few as 16 to perhaps 64K (K=1024). One simple way to express a central objective of this research is to understand how to achieve a situation in which "The more processors, the more performance."

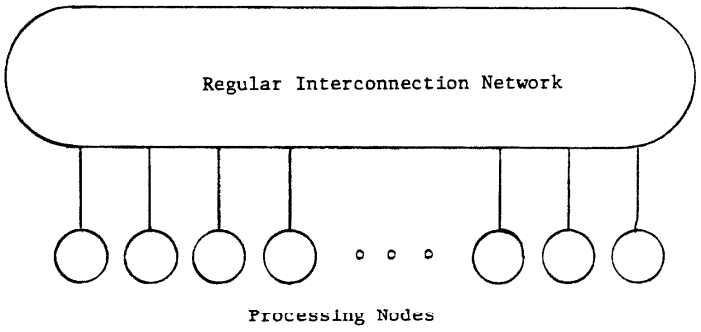


Figure 1-1

Hardware Model for a Homogeneous Machine

These architectures will be regarded as relatively "general purpose" computers in that (1) they are programmable, and in a style to be presented which makes no demands that the programmer know anything about the communication structure of the particular homogeneous machine executing the code, and (2) the homogeneous machine is no more specialized to a specific set of problems than other "general purpose" computers, but provides a high level of performance to all concurrent programming problems, including those for which the algorithms have no regular or fixed pattern of communication.

Most of the extensively studied VLSI architectures are special purpose systems, perhaps because performance on chips is very sensitive to the communication plan of the chip [Sutherland77], and the communication characteristics of a system are more difficult to generalize than, for example, its operation set. While special purpose machines can always be constructed to solve specific problems faster and more economically than general purpose computers, one must expect that there will always remain a class of applications that are either so unstructured that they are not suited to a rigid hardware structure, or for which there is insufficient demand to justify the design and construction of a special purpose machine.

The term unstructured is used here to describe concurrent programs or algorithms for which the graph of communication between the elements of the computation either varies with input data or for which no regular pattern can be discerned. Many problems are of this type. These problems may have irregular and dynamically changing communication graphs and require a machine and programming notation that permit irregular and variable communication.

Structured problems, such as matrix manipulation, signal processing, sorting and others where the communication graph is known before hand, adapt well to a homogeneous machine. However, for these problems special purpose machines can be designed which will out perform a general purpose homogeneous machine. In this thesis, the term general purpose will include applicability to problems for which the logical communication graph changes dynamically or is irregular or unknown.

The topology used to interconnect the parts of the machine is of great interest. It is clear that the communication capabilities of the network must be high to support the execution of interesting problems. The resulting structure must provide minimum communication delays yet must also be practical. Another important characteristic of the communication structure is expandability. The requirement that machine performance scale with the size of the system means that the hardware communication structure must not degrade or cost substantially more per processor as it is increased in its extent.

The hardware structure of the homogeneous machine is clearly motivated by certain characteristics of VLSI technology, as discussed in, for example, [Seitz82] and references cited there. Advances in the integrated circuit fabrication technology over the past 20 years and those anticipated create opportunities to build systems of greater complexity and switching speed at a dramatically lower cost per function. Thus systems of thousands of processors are not outrageous to contemplate. They are well within the capabilities of present technology. Replication is an intrinsic characteristic of VLSI technology, and is well exploited by the homogeneous machine. The cost in delay, area, and energy of long distance communication on and

between chips suggest that single processors will be fast and efficient only if relatively small. The opportunity for performance is in concurrency. The homogeneous machine satisfies these criteria very well also.

The advances in integrated circuit technology have created an opportunity for computer designers that must be accompanied by advances in concurrent programming techniques before they can be exploited, and vice versa. A number of concurrent programming notations have been proposed but have remained research toys, if only because they lack a suitable machine architecture on which they could be used. The underlying model of concurrency presumed by a programming notation must also be shared and supported by a concurrent hardware structure. Although it is in many circles just another "motherhood and apple pie" statement, this thesis treats the hardware and software together.

The von Neumman notion of a randomly accessible memory word as the basic unit of sequential machines and programs cannot survive in a homogeneous environment. The definition of a homogeneous machine makes the accessibility of the state of the machine a function of distance. The greater the distance between the need for a particular unit of information and the physical location of it, the greater the time required to obtain it. Locality in a homogeneous machine is achieved when it can be observed that the probability that two concurrent processes will communicate with each other decreases as the distance between them increases. A concurrent programming methodology must take this fact of life into account. In this thesis, the notion of the program "object" is used as the basic concept around which the programming model and the machine architecture are built. Objects are certainly not the only programming paradigm possible for

homogeneous machines, but are the model assumed for this thesis.

Objects and object-orient programming are derived from the concepts of Simula [Birtwhistle73] and Smalltalk [Ingalls78]. The object is an instance of a user defined data type. It contains local data which may be operated upon by the procedures defined for its particular type. The procedures are called attributes and their code may be shared by any objects of the defined type. Objects are referred to by the contents of reference variables which hold pointers. These pointers merely address the indicated object and identify its type; they do not indicate the physical location of the object.

If, as Backus [Backus78] suggests, programming can be liberated from the von Neumann style, the hardware structures that are constructed to support these new styles must certainly avoid the von Neumann bottleneck. This bottleneck is the narrow pipe through which all memory accesses must flow in conventional machines. This bottleneck is present, and is even more choked, in machines with multiple processors connected to a single memory. The choking may be somewhat relieved by increasing the cost and complexity of the system with such techniques as the interleaving of memory, crossbar switches and other "stunt" boxes [Thornton70], but the effectiveness of these techniques is necessarily limited by space, time, and cost considerations.

If the von Neumann bottleneck is to be removed, then its large, monolithic address space must become distributed among the various processors of a system. Also, the semantics of an "address" will grow beyond its current meaning which defines it as a fixed size word in a very large set of words. In an object oriented homogeneous machine, the address becomes a reference variable or pointer referring to an object. Objects are the basic

units from which data structures are built. Procedural attributes are defined for classes of objects and become a set of operations that manipulate the data contained in an instance of an object. These concepts are found in Simula and other languages and have been extended to operating systems by Hydra [Jones73]. In a system consisting of numerous processors each with their own "object memory", all access to an object within the memory of a particular processor is controlled by that processor. Objects communicate only by passing messages to other objects for which they contain a pointer.

Objects are distributed among the processors of a homogeneous machine and may be moved between them at any time to preserve the locality of communication. Objects are constrained to fit wholly within any given processor. They may execute concurrently where provided for by the programmer and where the opportunity exists. Objects may create other objects but cannot explicitly destroy other objects. Reference variables may be overwritten, copied and sent to other objects in messages. These operations result in a dynamic system where both the positions of the objects and the topology of their pointers change continuously. There can be no restrictions on the kinds of structures that might be generated (e.g. cycles in the pointers must be permitted). There is no way to enforce such restrictions nor is there any desire to. A garbage collection facility is required by an object oriented language to identify and eliminate inaccessible objects.

The object concept is powerful enough to have a broader interpretation, one that allows it to provide many of the facilities required to build complete systems. The files and directories in the file system of a machine running

UNIX[1] [Ritchie74] may be thought of as objects of whose type is implied by the operating system. If a number of such machines are connected via a network and links are permitted in a machine pointing at files or directories in another machine, essentially the same object-oriented situation can be seen to exist. Distributed database systems [Yu81] permitting multiple, concurrent access have a similar need to resolve the status of objects residing in multiple machines.

To meet the requirement that performance scale with system size, not only must the hardware communication facilities be suitable but software components, such as, message handling, garbage collection and resource allocation must also avoid algorithms and techniques that degrade as the system grows. The process of garbage collection is of great concern since the determination of whether or not a given object is referenced anywhere in the system is a global question, and was studied extensively in this research.

A number of principles of implementation might now be stated to provide the reader with a concrete picture of an object oriented programming system for a homogeneous machine.

Each processor contains its own memory to which it has exclusive access. There is no shared memory. Each processor/memory node runs its own copy of an operating system, which may be better thought of as a run time system. This code is always resident in the processor's memory and serves to support a particular programming environment that pervades the entire system of processors.

The memory associated with each processor may include disk storage. If mass storage peripherals are present on a processor, they can be regarded

[1] UNIX is a Trademark of Bell Laboratories.

as logically part of the processor's memory. No distinction is drawn in this thesis between objects stored in random access memory and those stored on disks if they are under the control of the same processor.

To permit the migration of programs from small machines to a large ensemble of machines will require a programming language common to both. Both the development and the execution of programs for small problems may occur on small, single processor machines. If large ensembles of processors are to be used to increase the preformance of such programs, the language used must be supported on both types of machines. Where they exist, concurrent programming languages perform poorly on single sequential machines. Sequential languages, by definition, are unable to take advantage of the concurrency available in a homogeneous machine. A programming model must be found that is suitable to both environments.

1.2. Related Efforts

A number of special purpose ensemble machines exist and many others have been proposed [Kung78,Browning80,Seitz82]. The unit replicated in these structures is typically a very small machine, either not programmable or with so little program storage that one could not include a run-time system to distribute work across the machine in execution. All such decisions must be made by the programmer and/or compiler in advance of execution. Machines of this type are highly specialized to and effective for specific regular problems, such as, the manipulation of large matrices or solutions to various graph problems.

The Torus [Martin81] and the Homogeneous Machine of [Locanthi80], incorporate more complex processors have achieve a correspondingly greater degree of generality. A concurrent programming notation, after the

style of Hoare's CSP [Hoare78], is proposed for the Torus machine and a functional subset of LISP serves as the basis for Locanthi's machine. These machines are general purpose to the extent that their programming notations are suitable for application to various problems. In both of these machines, the communication between the processors is restricted to a particular topology, hence the class of programs permitted by these machines and their programming notation is likewise restricted. Specifically, these two machines permit tree-like computation graphs. Computations occur in the leaves of the hierarchy and the concurrent components of the computation are created and destroyed at the leaves, expanding and contracting the logical graph of the computation.

The Actor model of programming bears many similarities to object oriented programming and the Apiary machine of Hewitt [Hewitt80] is centered about these concepts. The Apiary machine is a toroidal mesh of processing elements that are each host to a number of Actors. The goals of the Apiary machine and the programming model of Actors are directed at a general class of artificial intelligence problems.

1.3. Scope and Outline

This thesis presents the ingredients necessary to a homogeneous machine and an object oriented programming environment. Chapter 2 presents a programming model centered about object oriented programming. Extensions and restrictions of existing object oriented languages are presented which permit a hardware structure to support a message passing programming model with concurrency. Chapter 3 presents a new algorithm which enables the system to recover the resources occupied by inaccessible objects. The algorithm is shown to scale satisfactorily with

the size of the system. Chapter 4 addresses the requirements of the underlying hardware structure. Various interconnection strategies are evaluated with respect to their performance and cost, with the conclusion that a Boolean N-cube is attractive for the numbers of processors that might be used. Both the garbage collection algorithm and the interconnection topologies are evaluated by detailed simulation. In Chapter 5, topics related to the support of object communication and preserving object locality are addressed.

1.4. Conclusion

The conclusion of the thesis supports the contention that a highly concurrent programming environment can be implemented in a homogeneous machine. The use of a Boolean N-cube interconnection, the program object metaphor and a distributed, on-the-fly garbage collection algorithm provide a system which will have a level of performance that is proportional to its size. Machines of very large sizes may be built to solve large problems where concurrency is part of the solution program. The general purpose nature of object oriented programming makes this type of machine as generally useful as conventional single processor machines. Its ability to provide more performance by adding more hardware and its low per part cost due to its homogeneity make it a very good candidate as a VLSI architecture in years to come.

Chapter 2

Concurrent, Object-Oriented Programming

2.1. Introduction

This chapter presents a programming metaphor suitable for use with a highly concurrent machine architecture. This programming construct is derived from the Simula class concept [Birtwhistle73] and is related to similar concepts in other programming languages such as CLU [Liskov77], Smalltalk [Ingalls78] and ALPHARD [Wulf76]. Many of the concepts of object oriented programming have appeared in other languages, such as in the Actor System [Clinger81].

The machine architecture presented in succeeding chapters exploits the concurrency expressed in programs by managing the execution of objects on a collection of connected processors. The physical structure and organization of the machine are transparent to the objects. Several interconnection strategies may be used, however, some strategies will perform better than others. The structure's appropriateness to applications is evident only in its performance. Extensions to the physical structure can be made indefinitely without modifications to the existing parts or to the programming model of the machine seen by its users. Such expansions result in increased performance for concurrent programs.

Other methods have been found to exploit concurrency. These methods generally fall into two categories. First, machines have been presented

which are designed to solve particular classes of problems very well. The concurrency achievable in these machines is inherent in the algorithm or is explicitly programmed by the user. Examples of such machines are systolic arrays [Kung78] and the tree machine [Browning80]. Both of these machines are well oriented to matrix operations. Sorting, searching and graph problems have also been adapted to the tree machine. Attention to the details of the rigid hardware architecture make programming these machines difficult. Moreover, problems that perform well at one particular size may not work at all if the size of the problem is increased.

The second category is what may be called reduction machines [Berkling75]. The primary means of partitioning problems into concurrent parts is the separate evaluation of the parameters of procedure calls. These machines are typically directed at a particular language. The homogeneous machine described by [Locanthi80] is based on LISP and derives much of its advantage by concurrent evaluation of the CAR and CDR of LISP expressions. The machine presented in [Mago79] operates in a simpler but somewhat analogous manner without the caching of LISP nodes proposed in the Locanthi's machine. A reduction machine incorporating data flow concepts has been proposed [Treleaven80].

The Torus machine [Martin81] makes use of a twisted toroid to support the concurrent execution of procedure calls. The two procedures each run in a neighboring processor and may themselves initiate additional concurrent behavior. When a procedure returns its result it terminates and is destroyed. A tree of concurrently executing procedures is mapped onto the surface of a torus. A twist is applied to the torus in each dimension to increase the duty cycle of the individual processors. The Torus machine is a

member of a rare set of novel architectures that have been implemented.

The data-flow concept bears a close relationship to object oriented programming. The operators and the topology of token flow between them is fixed in static data-flow programs. The primary difference between object-oriented and data-flow programming is that objects, which may be thought of as operators, may be created and disposed of at a high rate and the communication between them is of a dynamic topology, while in a static data-flow machine operators are typically long lived and communicate in a fixed pattern. Machines proposed to execute data-flow programs such as [Dennis74] and [Davis78] have an explicit concept of message passing between program elements. In some cases, such as [Arvind81] the data-flow machines exhibit some similarities with reduction machines where recursive procedures have been implemented.

Computer programs have traditionally been treated as single, albeit large, sequential machines [Backus78]. Most programming languages such as FORTRAN, ALGOL and APL permit only a strictly sequential specification of algorithms. This development is a natural one since the machines on which such programs are run provide no additional benefits for program specifications containing potential concurrency. Programming languages have been available since the 1960s that allow the user to indicate concurrency in programs (Simula 67), and since then others have come into being such as Concurrent Pascal [Brinch Hansen75]. Many ideas have been offered to programmers to provide them with notations that include the concept of concurrency such as in CSP [Hoare78]. As a rule, these languages have addressed only the notational and semantic issues and have ignored any notion of locality among program components.

In several cases concurrent programming languages have presumed a particular hardware environment where two or more processors have equal access to a single memory. A number of machines of this type exist on the market, which may explain some of the attractiveness of this environment. However, we consider here a broader situation where the time required to access data in the system is a function of the physical distance to the data. This relationship effectively prevents the use of techniques such as shared variables to implement program constructs like semaphores. Any type of global variable is prohibited. To decrease the communication delay between various objects, the distance between those objects will be made small. Maintaining locality by moving data is possible only where the data is part of an object and the object can be moved *en masse* to increase its ability to communicate with other objects. Moving data to increase locality is not possible without the grouping of data and program code into neat, independent bundles like objects. With the object concept the user must partition programs into bundles of code and data providing the homogeneous machine with the ability to retain locality among the objects at run time. Without the preservation of locality, communication between program components becomes so excessive that comparatively little computation is done.

Ideally, the languages in which programs are written should make the specification of locality both convenient for the programmer and a natural part of the language syntax. Block structured languages such as Algol, Pascal and PL/1 approach this goal by imposing scoping rules on the programmer. Simula and Smalltalk go one more step and provide a convenient means to associate data with program. The object construction of these languages not only brings together related pieces of data but also

identifies the program code which is to operate on that data.

2.2. Overview of Simula

Simula is a derivative of ALGOL 60, and as the name implies, is useful in programming simulation problems. It is also a general purpose language in which a large body of code has been written for computer aided design, system support and graphics applications. Simula uses a superset of ALGOL 60 syntax with several important additions. Program data in Simula can reside either on a stack, as in ALGOL, or in a dynamic memory area or heap. The data items that reside in the heap are termed objects and are instances of data type definitions. As references between and among objects are changed under program control, a garbage collection procedure is used to remove inaccessible objects and to compact the heap.

In this section we present some of the major features of Simula. A familiarity with the ALGOL programming language is assumed. A complete description of Simula can be found in [Birtwhistle73].

The Simula object is the basic data abstraction mechanism of the language. Objects are defined by CLASS declarations which are written by the programmer to describe an entity that consists of local data and a set of procedures. The procedures of the CLASS that are accessible to other objects are called attributes and may be regarded as a set of operations defined on objects of the CLASS. The name of the CLASS is regarded as the name of a data type to which all instances of the CLASS belong. Objects are instances of a CLASS declaration and are created explicitly with the NEW statement. The local data is unique to each instance of the CLASS. The procedure attributes of an object are code that is shared among all instances of the object's CLASS. The reference variable is an atomic data type that can

hold a pointer to an object. The name of the reference variable is used to indicate the object to which the variable currently refers and is the means by which the remote attributes of the object are invoked.

Simula supports a set of atomic data types consisting of integer, Boolean, real and character variables. A representation for text and operations on text are also included. In addition, arrays of the atomic data types may be specified. User defined data types are termed CLASSes by Simula and consist of a collection of variables and procedures. A reference variable is a pointer that refers to a specific instance of a class. Reference variables are declared to be of a specific type and can only contain pointers to instances of a particular class or they may contain the null pointer (NONE). The operator ":-" is used in Simula to assign a reference variable a pointer. Objects may contain reference variables to other objects permitting arbitrary data structures to be constructed. Listed below is a simple Simula program.

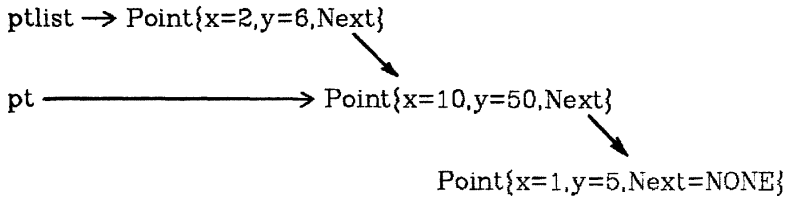
```
BEGIN

    CLASS Point(x,y); REAL x,y;
    BEGIN
      REF(Point) Next;

      REF(Point) PROCEDURE Copy;
      Copy :- NEW Point(x,y);

      REF(Point) PROCEDURE Scale(r); REAL r;
      BEGIN
          x := x*r;
          y := y*r;
          Scale :- THIS Point;
      END;
    END of CLASS Point;

    REF(Point) pt,ptlist;

    pt :- NEW Point(1,5);
    pt.Next :- pt.Copy;
    ptlist :- NEW Point(2,6);
    ptlist.next :- pt.Scale(10);
END;
```

In this program, class point is defined with 5 attributes. The real variables x and y hold the coordinates of the point. The reference variable Next may hold a pointer to some other point so that a linked list of point objects may be constructed. The procedure attributes Copy and Scale define operations on point objects. Attributes of classes are invoked by using the name of a reference variable followed by the name of the attribute separated by a dot ("."). Procedure attributes may or may not require parameters, and they may return a value as shown here.

Within the block defining class point, procedures and their code may reference data attributes as local variables as seen in procedure Scale. Outside of this block, the scope rules make all the declarations of data and procedures with class point invisible except as attributes of point objects.

Instances of objects are created using the NEW statement. This statement allocates space for the object in the heap and returns as its value

a reference variable pointing to it. In the program, the reference variable pt is assigned the value of a NEW statement. The attribute Next of the newly created point is then assigned a pointer to another new point created using the Copy attribute defined for point objects. At the end of the program the data structure is as follows.



Classes can be defined with any attributes the programmer determines are necessary. Instances of objects can be made and manipulated to perform any task. An important characteristic of programs written in this manner is that the code that operates on the data of a class instance is identifiable and modular. The procedures Copy and Scale in the example are shared by all instances of point objects. Thus, the existence of a point object implies the need for the specific code associated with class point.

The type checking of Simula is strong, that is, all variables and procedures are assigned a type and only legal operations are permitted between various types, whether user defined or built-in. Simula permits the relaxation of type checking with two features, subclasses and virtual procedures.

Subclasses allow a hierarchical grouping of classes and permit the definition of one class to inherit attributes from the definition of another class. A justification for subclasses is in [Ingalls78]. In the following example class "person" is defined. Classes "man" and "woman" are defined as subclasses of "person".

```
BEGIN

    CLASS Person(Name); TEXT Name;
    BEGIN
        ! Attributes of Person ;
    END of CLASS Person;

    Person CLASS Man;
    BEGIN
        ! Attributes Unique to Man ;
    END of CLASS Man;

    Person CLASS Woman;
    BEGIN
        ! Attributes Unique to Woman ;
    END of CLASS Woman;

END;
```

Instances of all three classes are regarded as instances of "Person" in type checking. Therefore, a reference variable of type "Person" may contain a pointer to objects of any of these three classes. Also, any attributes defined in "Person" are also part of "Man" and "Women". Where conflicts arise in the names of attributes, the declarations of the subclass override those of the superclass. The scope of the code in a subclass includes the corresponding block level in the superclass. That is, the procedures and data declared in the attributes of "Person" are visible to the code in the "Man" and "Women" definitions.

When a class hierarchy is constructed, the attributes have fixed meanings using the subclass mechanism shown in the example. It may be desirable to permit the definition of a subclass to redefine attributes that would otherwise be inherited from its superclass. It may also be desired to give the superclass access to the attributes of a subclass, based on the exact class membership of an object at run-time. The Simula VIRTUAL mechanism provides these abilities as another method of relaxing the otherwise rigid typing and scoping rules.

Briefly, the virtual mechanism consists of declaring one or more attributes of a class to be VIRTUAL but without actually defining the attribute. Subclasses may define or redefine the attribute name, though such definitions and declarations must match the type of the VIRTUAL attribute. When a virtual attribute of an object is invoked, the attribute definition of the lowest subclass to which the object belongs is used. This is in contrast to using the class of the reference variable to determine which set of attributes are applicable.

2.3. Extensions for Concurrency

Simula objects are modular with regard to both their data and their code. The user definitions of Simula CLASSes partition data with the procedures that will modify the data, making objects that are modular and independent except where they interact with other objects. The dot operator (".") is used to invoke the attributes of objects in Simula. When an object attribute is invoked a message is sent to the object containing any parameters required by the attribute.

```
pt.Scale(R);
```

This Simula statement can be interpreted as the sending of a message to the object referred to by pt containing the name "Scale" and the value of the parameter R. The analogous operation in Smalltalk is actually presented by Ingalls as message passing. If objects can then stand alone with their own code and data and have the ability to send and receive messages from other objects, they can execute concurrently.

Simula comes equipped with its own facilities for concurrency. These are the RESUME and DETACH statements and are used, in conjunction with other Simula features, to provide the simulation facilities for which Simula

was originally intended. These two commands implement a context switch between Simula objects. Objects each have their own stack and "program counter", thus permitting program control to be transferred from one to another just as jobs are multiplexed in a time shared system. The RESUME command suspends the execution of the current object and resumes execution of a named object from where it was last suspended. The DETACH command suspends the current object and resumes the main program. These features provide no synchronization mechanisms and require very detailed attention by the programmer. Also, locality and modularity are degraded by these facilities. It is clear that the intention of these features was to support simulation and not concurrent programming. Simula does provide each object instance with its own stack so that the execution of each object may utilize recursion and block structuring as would any Algol-like program. While the RESUME and DETACH facilities may be unsuitable for the purposes here, concurrent execution does require that each object contain, as part of its state, a stack of display records.

Objects communicate by invoking the attributes of other objects. To permit objects to be located in physically separate processors, the action of invoking an attribute in an object and receiving a result must be implemented by message passing. All communication between objects is via a message passing facility. The hardware facilities and the run time system of the homogeneous machine implement the passing of messages between objects. The following characteristics of this message passing system are assumed.

(1) Messages sent concurrently by different source objects but intended for the same destination object arrive at the destination in an arbitrary

order.

(2) Successive messages sent from a particular source object to the same destination object arrive at the destination object in the order they are sent.

(3) Messages received by an object are received whole. Any assembly of pieces of the message is transparent to the recipient.

(4) All messages will eventually reach their destination and need never be retransmitted. All error handling is transparent to the objects.

(5) Messages received by an object are assured of being intended for that object. An object will not receive messages not intended for it.

(6) The only prerequisite needed to send a message to an object is that the sender have a reference variable identifying the object.

Objects are self contained and may execute concurrently. Given independent objects which communicate via a message passing mechanism, we propose a model of concurrent programming for the object-oriented environment. The extensions take the form of conventions placed on the passing of message between objects. Means are introduced whereby objects may execute concurrently, as well as synchronize with the completion of the activities of other objects. The original semantics of Simula remain in effect, except where the added conventions of message passing are felt. In a succeeding section restrictions are placed on the language to prevent the expression of programs that would require global communication.

The activation of an object attribute sends a message to the object and may initiate concurrent execution between the sender and the receiver. The following program segment will cause the object executing this code and the vector object pointed to by Vec to execute concurrently.

```
Vec :- <expression returning a pointer to a vector>;
  .
  .
  .
Vec.sort;
a := b + c;
```

The syntax of Simula indicates that no result is expected for the statement "Vec.sort" and the execution of the sort operation by the vector object may proceed independent of the requester. Thus, the requester will go on to execute the assignment of "a" while the vector object sorts its elements. In the program segment below, a result is expected by the requester.

```
Vec :- <expression returning a pointer to a vector>;
    .
    .
    .
c := Vec.elemt(i);
a := b + c;
```

In this example, the syntax indicates that the value of an object attribute is required by the requester. The requesting object is then made to wait until the result is sent in a response before executing the assignment of the result to "c" and then the assignment of "a". If "Vec.elemt(i)" is present in any context where a result is expected, such as in the assignment above, or as a parameter or in an expression, the object making the request of the vector will stop all execution until the vector has responded with a value. When the response has arrived, the requester has then been synchronized with the activities of the vector object.

The termination of an attribute at a destination object may send a result which is received by the original sender as the value of the attribute. An object attribute returns a result using the syntax and semantics of returning a value from a procedure. There is, however, one difference. As shown above, the requesting object may not require a result even though the

attribute requested may be defined to return a result. The syntax and semantics of Simula indicate whether or not one is required by the requester and the requester makes this known to the destination object in its message. Thus, if the requester's message indicates that no result is required, then the destination object will not send a result regardless of the destination object's definition.

The messages received by an object are acted upon one at a time. This restriction is severe, and is a different choice than was made for the Actor model [Clinger81], but permits a simplified programming style where many of the details of concurrent programming are hidden.

The restriction that only one message may be acted upon at a time provides several important characteristics for the objects. First, it insures that the data of an object is "guarded". That is, the code that might modify it is a "critical region" and there is "mutual exclusion" among those attributes that might change it. The concepts of critical regions and mutual exclusion are synchronizations that must be available in concurrent programs to insure a particular behavior by the program. Permitting only one attribute of an object to be executed at a time prevents the modification of the object's state by another attribute in ways that programmer did not specify.

Messages are queued for each object attribute in the order in which they arrive. When the object has finished executing the attribute associated with one message, it may then begin the execution of the actions associated with the next message in any of the object's attribute queues. The selection of the attribute queue from which to take a message is arbitrary unless the selection is controlled by the specification of the object's class. If all the

attribute queues are empty, the object waits in an idle state for a message. A message would typically contain a set of actual parameters supplied by another object which holds a reference variable identifying the object in question. If the originator of the message and the definition of the attribute require a result to be sent to the originator, the transmission of the result occurs when the execution of the attribute terminates.

The transmission of a message to an object is caused by the activation of an attribute of an object referred to by a reference variable. The transmission of a result is caused by the termination of the attribute of the object where a result is defined for the attribute and required by the originator. The syntax of these operations is the same as that used in conventional Simula for the invocation of attributes and for returning values from procedures. This is in contrast to the CSP notation [Hoare78] where the "?" and "!" operators are used explicitly to receive and transmit from and to named communication channels. The channels between CSP program components must be declared and both the sender and receiver must be executing the output and input operators, respectively, for the communication to take place.

One new language feature is added to Simula to permit the programmer to control the otherwise arbitrary selection of attribute queues. The SELECT statement is made part of an attribute declaration where selection of messages from the associated message queue are to be conditional. If no SELECT statement is present in the attribute declaration, the attribute may be arbitrarily selected whenever an attribute terminates and there is a message waiting in the attribute's queue. If present, the SELECT statement takes a Boolean expression as an argument. The attribute is left out of the

arbitrary selection when the value of the expression is false. An arbitrary selection is made among those attributes for which there are messages and whose SELECT expression evaluates as true, or for which there is no SELECT statement. The Boolean expression may contain any of the relational and logical operations of the language and may contain references to any of the object's local data items. In addition, the expression may use the names of attributes. Where the names of attributes appear in the expression, they have a value of true when the message queue of the named attribute is non-empty, the value is false if the queue is empty. Examples of the SELECT statement can be seen in the Gaussian elimination example in a succeeding section. This statement is the only feature added to the syntax of Simula for the purposes of concurrent programming.

The restriction that only one message may be removed from the attribute queues and acted upon at a time provides a synchronization with other objects. In addition, due to property (2) of the message passing system, an object sending a message to another object is assured that all other messages that have been sent in advance will be received, and further, all other messages invoking the same attribute will have been acted upon when it receives a response to its message. This use of a FIFO as a message queue for each attribute is a means of controlling the non-determinism introduced by the arbitrary ordering of the message passing system. The selection of which attribute is to be executed when there are messages waiting in several attribute queues may be controlled by the programmer with the SELECT statement where a completely arbitrary choice is undesirable.

Objects may activate tasks in other objects without suspending their own execution. Synchronization between concurrently executing objects takes place when the originator again sends a message to the "slave" object. Since the "slave" object must finish the previous, selected task before acting on the new message, any response by the "slave" object indicates the completion of all previous tasks as selected by the "slave" object. Of course, the originating object may wait on an initial response from the object, thus precluding any opportunity for concurrency, or having once initiated a task in another object, the originator may never synchronize with the "slave" but may indirectly cause a third party to do so.

To preserve the sequence of operations programmed in the objects, one proviso must be added to the initiation of concurrent behavior. While an object sending a message to another object for which no response is indicated need not wait for execution of the attribute, it is necessary that the message be put in the attribute queue of the destination object before another message is sent by the originator. Property (2) of the message passing system is intended to assure, however implemented, that any messages that might be sent to the destination object as a result of further execution by the originator will be acted upon after the first message or under control of the attribute selection logic. Thus, the sequence of operations as specified by the programmer is preserved.

An example of these types of interactions between objects is often found in an object containing a vector (or array) of other objects. After loading the vector object with references to various other objects, the controlling object or an object to which a reference to the vector has been passed may instruct the vector to sort itself. Since this operation does not require a result to be

produced immediately, the controlling object may go on about its business until it requires an element of the sorted vector. Any object requesting a particular element of the vector will be made to wait until the vector has completed the sort, if in the vector object a SELECT statement is used to prevent the selection of requests for elements when a sort request is present. Any requests for elements made after a sort request is placed in the sort attribute queue will not be executed until after the vector is sorted. At the completion of the sort the vector object will service a new message from its attribute queues. By sending the a response for an element request containing the desired element, the requesting object is released for further execution. In this way, any number of objects which hold handles referring to the vector are synchronized with the completion of the sort.

Of course, no guarantees are made as to which of several messages from independent objects will arrive first at the vector object. Non-deterministic behavior, resulting from races between objects, is certainly possible. The programmer is responsible for insuring that objects which expect the vector to be sorted do not make requests before the vector begins the sort operation. The programmer is assisted in maintaining this sequencing by using SELECT to assure that all execution following the *transmission* of the sort message to the vector will find the vector to have been sorted.

The opportunities for concurrency presented here are explicitly programmed. Additional implicit concurrency can be had if objects that would otherwise wait for the response to a message continue execution until the response is actually required by the program code. In some cases, the parameters to procedure calls could be evaluated concurrently since procedures cannot be entered until all of the actual parameters are present.

These types of optimizations by the compiler or interpreters are not expected to be a major factor in the benefits provided by this type of distributed architecture but are available to compiler writers. The "futures" construct of the Actor system [Hewitt77] is based on this idea.

To add concurrency to an object-oriented language we have merely changed the semantics of invoking an object attribute. Instead of calling a procedure defined in the object's class definition, we send the object a message. If a value is required from the attribute, execution is suspended until it is available. This situation will appear the same as if a procedure had been called in a conventional language. If no response is required, execution continues. The object at the destination may continue or begin execution as well. Synchronization between the objects is accomplished by the message passing mechanism. The attribute FIFOs or queues insure mutual exclusion between the various attributes of the object.

2.4. Restrictions to Limit Global Communication

To enforce a degree of locality among the objects and data of a program, some modifications to the scoping rules and construction of programs are required.

ALGOL scoping rules permit program code to access any data declared in any textually enclosing block. In Simula, for instance, data declared at the highest level are visible and can be manipulated by all code in the program, including the code internal to class definitions. Such global variables cannot be permitted in a distributed machine of the type discussed here. Not only would their access require excessive communication but they would be lacking the synchronization required to support reliable concurrent behavior. Without synchronization between the concurrent objects,

modifications and accesses to global variables would have no controlled sequence making deterministic program behavior difficult if not impossible. Therefore global variables cannot be permitted.

The message passing model for objects requires that all data related to the execution of an object be either part of its internal state or arrive as part of a message. This model does not allow for the direct access to global variables provided in conventional languages. It also does not allow the internal state or data attributes of objects to be directly accessed as permitted by Simula. As a result, only procedural attributes of objects are accessible to other objects and all access and modifications to the data attributes must be accomplished using the procedural attributes.

To avoid global variables and other nonlocal access to data, the following restriction is made. Variable declarations may be made only within the body of a CLASS declaration. Further, variables declared are visible only within the immediately enclosing CLASS declaration. In other words, variables are declared only within objects and may be accessed directly only by the code defined as part of that object.

The passing of parameters to object attributes must likewise be restricted. To provide the destination object with the data required by its attributes, all parameters to attributes must be passed by value. Passing parameters by reference and by name would result in a loss of locality, where the data required by the object could reside at some other location in the system. If, for example, an array is a parameter to an attribute, the contents of the array must be incorporated in the message to the object attribute. In the case of reference variables, the value of the reference variable is transmitted, however, the value of a reference variable is the

actual pointer to an object. The restriction that parameters be passed by value thus permits the passing of pointers to objects in messages.

Class definitions can be nested to any depth, and normal scoping rules apply to the class names. The scoping of data does not permit access across class boundaries. The definition of procedures can cross class boundaries only if the procedure restricts its access to its own local variables and parameters. Procedures that manipulate internal class data cannot be called from other classes as these accesses would have same problems as global variables. Of course, the definition of a subclass may access all the data and procedures defined in the superclass as though the code of the subclass were actually within the body of the superclass.

Instances of objects can be seen to be a subset of the concept of a distributed process [Brinch Hansen78]. Objects may execute concurrently as do distributed processes and they communicate via message passing. Distributed processes communicate and synchronize by procedure calls and guarded regions. The message passing of objects is made to appear as procedure calls and the attributes of the objects are mutually exclusive, implementing a set of guarded regions for each object. Like an object, a distributed process may access only its own variables. Unlike distributed processes, objects cannot be interleaved, meaning that the attributes of an object cannot be used concurrently as can external requests of distributed processes. Objects are less general in nature than distributed processes with the intention that objects be better suited to execution in a homogeneous machine.

To allow objects to be distributed among many processors and their memories, we have restricted their definition such that all access to an

object by other objects is exclusively through a message passing facility. Direct access to an object's internal data is therefore prevented. An object may manipulate its own data or it may send messages to other objects to indirectly affect their data. If one or more variables must be "shared" by a number of objects, then the variables may themselves be contained in an object and a pointer to the object can be given to all those objects that require access to the variables. In this way, all the techniques of sequential languages can yet be used but are made to fit within the distinct boundaries of objects. In exchange for these restrictions, concurrency is made available and can be taken advantage of without explicitly specifying critical regions, semaphores or other types of synchronization mechanisms.

2.5. Concurrent Programming Examples

To illustrate the usefulness of the modified version of Simula described above, several example concurrent programs are presented.

2.5.1. Two Dimensional Shapes Clipping

One of the most common tasks of interactive graphics programs is the display of two dimensional shapes on a plotting device. In the process of displaying shapes, the shapes must be transformed and clipped to fit the space and coordinate system of the plotter. The clipping of shapes is the process of removing those parts of the figures that fall outside of the plotting window. The plotting window is a rectangle denoted by upper and lower bounds in the Y dimension and right and left bounds in the X dimension.

In this example, the task of clipping is pipelined into four concurrent objects. Various sequential algorithms exist to perform fast clipping [Newman79]. These algorithms have been highly optimized for single

sequential machines. In the example here, a brute force technique is used where each of four objects clips line segments against one of the four sides of the window. Each of the four objects performs a simpler task than the single sequential methods, and since the four objects are pipelined and running concurrently, the effective rate at which line segments are clipped is greater than the sequential method, if other factors remain equal.

In the example, a Window object is defined. The initialization attribute (Init) constructs a linear list of four Clipper objects. At the end of the list is a pointer to the plotting device which is not defined here. It is assumed that the plotting device has a attribute which plots the line segments. When the Clip attribute of the window object is invoked by another object holding a pointer to the window, a list of line segments is sent to the window which the window then puts through the pipeline of Clipper objects.

The Clipper objects each have their own Val variable which is the limit that they clip line segments to. They also may have a reference variable to either a plotting device or the next clipping object in the pipeline. In this example, only one of the clipping attributes are used in each of the four clipping objects. The window invokes ClipAbove in the first clipper, then the first clipper may invoke ClipBelow in the second, and so forth until the last clipper object invokes an attribute of the plotting device.

Each Clipper object receives line segments from its successor in the pipeline. It performs some simple tests to determine if the line segment is fully on one side or the other of its limit. If the line is totally outside of the window limit, then the clipper attribute terminates and will go on to process the next line segment. If the line segment is completely on the other side of the limit, then it passes it on intact to the next clipper object, or to the

plotter. If the line segment straddles the limit, then a new line segment is computed and sent on to the next clipper.

In every case where a line segment is sent via an attribute to a clipper object or to the plotting device, no response is called for, permitting concurrent execution of all the objects in the pipeline. Thus, line segments can be "pumped" through the pipe at a rate determined by the slowest of the objects.

```
CLASS Window;
BEGIN

    CLASS Clipper;
    BEGIN
      ! Data of Clipper Objects ;
      REAL Val;
      REF(Clipper)NextClip;   ! Next Clipper in Pipeline ;
      REF(Plotter)PlotDevice;

      ! Attributes of Clipper Objects ;

      REF(Clipper) PROCEDURE Init(InitialVal,InitialNext,InitialDev);
      REAL InitialVal; REF(Clipper)InitialNext; REF(Plotter)InitialDev;
      BEGIN
          Val := InitialVal;
          NextClip :- InitialNext;
          PlotDevice :- InitialDev;
          Init :- THIS Clipper;
      END of Init;

      PROCEDURE ClipAbove(X1,Y1,X2,Y2); REAL X1,Y1,X2,Y2;
      IF Y1>Val OR Y2>Val THEN BEGIN
          IF Y1>Val AND Y2>Val THEN NextClip.ClipBelow(X1,Y1,X2,Y2)
          ELSE BEGIN
             REAL NewX;
             NewX := X1+(Val-Y1)*(X2-X1)/(Y2-Y1)
             IF Y1>Y2 THEN NextClip.ClipBelow(X1,Y1,NewX,Val)
             ELSE NextClip.ClipBelow(X2,Y2,NewX,Val);
          END;
      END of ClipAbove;

      PROCEDURE ClipBelow(X1,Y1,X2,Y2); REAL X1,Y1,X2,Y2;
      IF Y1<Val OR Y2<Val THEN BEGIN
          IF Y1<Val AND Y2<Val THEN NextClip.ClipRight(X1,Y1,X2,Y2)
          ELSE BEGIN
             REAL NewX;
             NewX := X1+(Val-Y1)*(X2-X1)/(Y2-Y1)
             IF Y1<Y2 THEN NextClip.ClipRight(X1,Y1,NewX,Val)
             ELSE NextClip.ClipRight(X2,Y2,NewX,Val);
          END;
      END of ClipBelow;

      PROCEDURE ClipRight(X1,Y1,X2,Y2); REAL X1,Y1,X2,Y2;
      IF X1>Val OR X2>Val THEN BEGIN
          IF X1>Val AND X2>Val THEN NextClip.ClipLeft(X1,Y1,X2,Y2)
          ELSE BEGIN
             REAL NewY;
             NewY := Y1+(Val-X1)*(Y2-Y1)/(X2-X1)
             IF X1>X2 THEN NextClip.ClipLeft(X1,Y1,NewY,Val)
             ELSE NextClip.ClipLeft(X2,Y2,NewY,Val);
          END;
      END of ClipRight;
```

```
      PROCEDURE ClipLeft(X1,Y1,X2,Y2); REAL X1,Y1,X2,Y2;
      IF X1<Val OR X2<Val THEN BEGIN
           IF X1<Val AND X2<Val THEN PlotDev.Plot(X1,Y1,X2,Y2)
           ELSE BEGIN
              REAL NewY;
              NewY := Y1+(Val-X1)*(Y2-Y1)/(X2-X1)
              IF X1<X2 THEN PlotDev.Plot(X1,Y1,NewY,Val)
              ELSE PlotDev.Plot(X2,Y2,NewY,Val);
           END;
      END of ClipLeft;
   END of CLASS Clipper;


   ! Data and Attributes of Class Window ;

   REF(Clipper)Clp;   ! Reference to the head of the Pipeline ;

   REF(Window) PROCEDURE Init(PlottingDevice,RightLimit,LeftLimit,
                                          UpperLimit,LowerLimit);
   REF(Plotter)PlottingDevice;
   REAL RightLimit,LeftLimit,UpperLimit,LowerLimit;
   BEGIN
      ! Set up Pipeline of Clipper Objects ;
      Clp :- NEW Clipper.Init(RightLimit,NONE,PlottingDevice);
      Clp :- NEW Clipper.Init(LeftLimit,Clp,NONE);
      Clp :- NEW Clipper.Init(UpperBound,Clp,NONE);
      Clp :- NEW Clipper.Init(LowerBound,Clp,NONE);
      Init :- THIS Window;
   END of Init;

   PROCEDURE Clip(Vec); REF(Vector)Vec;
   BEGIN
      INTEGER I,Len;
      REF(Segment)Seg;
      Len := Vec.Length;
      ! Take a list of line segments and put then into the   ;
      ! Clipper Pipeline.  Actually, handing the list to the ;
      ! Clipper would be better but less instructive.        ;
      FOR I:=1 STEP 1 UNTIL Len DO BEGIN
           Seg :- Vec.Element(I);
           Clp.ClipAbove(Seg.X1,Seg.Y1,Seg.X2,Seg.Y2);
      END;
   END of Clip;

END of CLASS Window;
```

The example is not only conventional and simple in appearance but is also a valid sequential program. If concurrency were not available and the invocation of object attributes were implemented in normal Simula, this program would still work as indicated. The model of concurrency proposed

for Simula thus permits the concurrent specification of this algorithm with enough sequential constraints to insure deterministic behavior.

2.5.2. Correlation

Correlation of digital signals is an operation commonly performed in signal processing applications. In many cases, the intention is to find a measure of how similar two signals are to each other. The recognition of radar images and speech are typical applications. The correlation of two sequences, x and y for delay d is given by the equation below.

$$R_d = \frac{1}{N-d} \sum_{t=1}^{N-d} x_t y_{t+d}$$

Where:

N = *Number of samples in* x and y

$d = 0 \cdots m-2, m-1, m$

The sequence R is the correlation of x and y for delays from 0 to m. It is clear that each element of R can be computed independently and hence concurrently. If we wish to correlate some input signal x with and number of different y patterns, these computations may be done in parallel as well.

In the following example, we have several sequences y to be correlated with an input sequence x. For some m we wish to determine the sequence y with the greatest value of R_d. This result would indicate which sequence y best matches x and with what delay d.

A class SingleCorrelate is defined which will compute R_d for x and some y for a given d. An object of class MultiCorrelate creates a SingleCorrelate object for each sequence y in a list of sequences and for each delay value desired. The MultiCorrelate object retains no references to the SingleCorrelate objects but instead gives each SingleCorrelate object a

reference to itself. When the SingleCorrelate object has computed its R_d it invokes the Take attribute of the MultiCorrelate object to send its answer. The Take attribute tests the answer and records it if it is the largest received up to that point. It also decrements a counter which indicates how many answers are yet outstanding. When all answers have been received, the variable AnswerReady is set true to show that the answer is valid. A graph showing the object structure used in this computation is found in Figure 2-1.

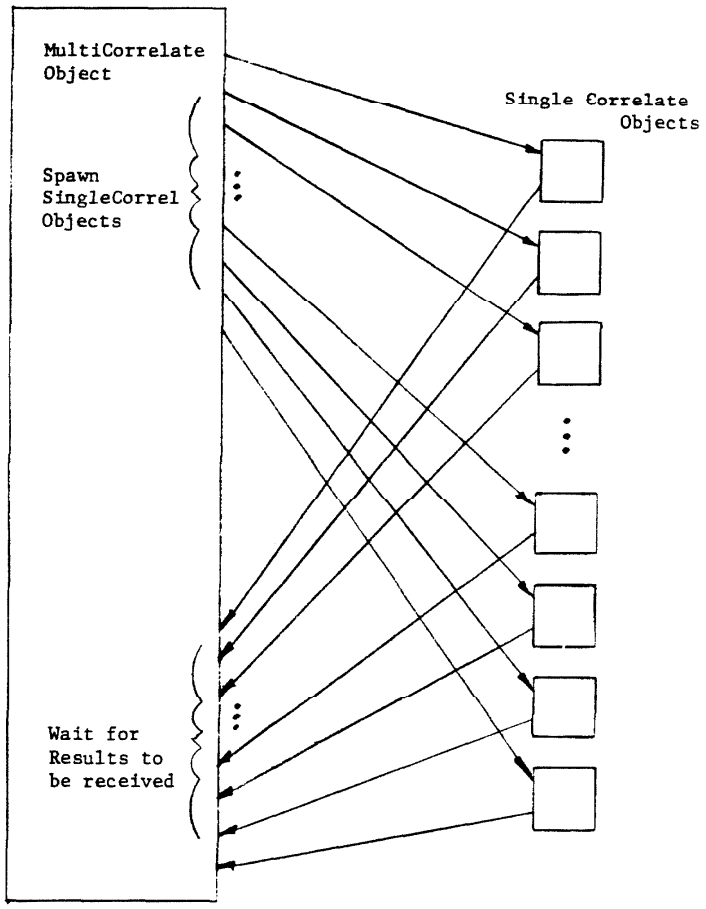


Figure 2-1

Object Structure for Correlation

```
CLASS MultiCorrelate;
BEGIN

    CLASS SingleCorrelate;
    BEGIN

      REAL ARRAY Y;
      INTEGER Delay;
      REF(MultiCorrelate)AnswerTo;

      REF(SingleCorrelate) PROCEDURE Init(RefSig,SendAnswer,InitDelay);
      REAL ARRAY RefSig; REF(MultiCorrelate)SendAnswer;
      BEGIN
          Y :- RefSig;
          AnswerTo :- SendAnswer;
          Delay := InitDelay;
          Init :- THIS SingleCorrelate;
      END of Init;

      PROCEDURE CorrelateWith(X);
      REAL ARRAY X;
      BEGIN
          REAL R;
          INTEGER I,L;
          L := X.Length-Delay;
          FOR I:=1 STEP 1 UNTIL L DO R := R + X[I] * Y[I+Delay];
          AnswerTo.Take(R/L);
      END of Correlate;
    END of CLASS SingleCorrelate;

    ! Data variables in which to hold the answer ;
    INTEGER SelectY,SelectDelay;
    REAL Answer;
    BOOLEAN AnswerReady;

    REF(MultiCorrelate) PROCEDURE CorrelateAll(X,ListOfY,M);
    REAL ARRAY X; REF(ArrayList)ListOfY; INTEGER M;
    BEGIN
      INTEGER I;
      Count := M * ListOfY.Length;
      Answer := -Infinity;
      AnswerReady := FALSE;
      FOR I:=1 STEP 1 UNTIL ListOfY.Length DO BEGIN
          FOR J:=1 STEP 1 UNTIL M DO
            NEW SingleCorrelate
                .Int(ListOfY.Element(I),THIS MultiCorrelate,J)
                  .CorrelateWith(X);
      END of FOR;
    END of CorrelateAll;

    PROCEDURE Take(R,M,WhichY); REAL SomeR; INTEGER M,WhichY;
    BEGIN
      Count := Count - 1;
```

```
        IF R>Answer THEN BEGIN
            Answer := R;
            SelectY := WhichY;
            SelectDelay := M;
        END;
        IF Count=0 THEN AnswerReady := TRUE;
    END of Take;
END of MultiCorrelate;
```

As in the previous example, this program also performs properly if executed sequentially. Here an object has been created for every basic loop of the correlation function. Such a technique could potentially create very large numbers of objects. If more objects are created than there are processors available, then the execution time of the problem will increase in proportion to the number of excess objects.

2.5.3. Gaussian Elimination

The last concurrent programming example is an implementation of a Gaussian elimination algorithm due to [Johnsson81]. In this algorithm, an array of computing elements is used to transform a banded matrix into an upper triangular matrix where all the lower triangular elements are zero. Other concurrent matrix algorithms have been proposed for direct implementation in hardware such as the systolic array algorithms of [Kung78]. This type of computation is akin to data flow machines where a fixed interconnection of computing elements operates on streams of input data. The computing elements used in these machines are typically small, containing arithmetic hardware such as multipliers and dividers and a number of bytes of state.

The computational array of [Johnsson81] goes on to provide the facilities necessary to solve a set of simultaneous equations. Figure 2-2 is a diagram of the complete array. The cells at the far left of the array and at the very

bottom produce the solution to the equations. The remaining portion of the array takes streams of matrix elements from above and from the right and produces the upper triangular matrix in the stack cells near the bottom. The elements labeled with a Z in the diagram denote registers.

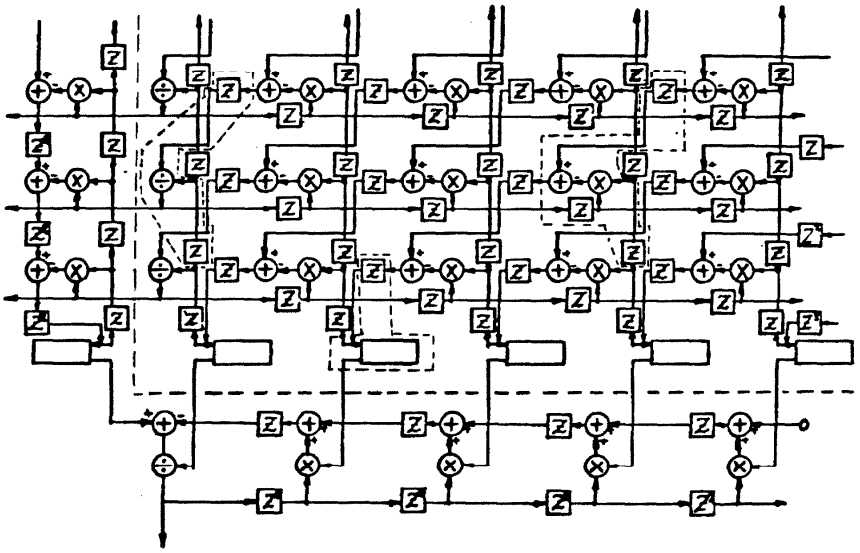


Figure 2-2

Computational Array for Gaussian Elimination

In the following example program, we implement the Gaussian elimination portion of the array. This problem differs substantially from the foregoing examples because the concurrent objects must accept not one stream of input messages but several and yet maintain their synchronization. The SELECT statement is used to cause the objects to accept the input of the specific data elements only when that data element is lacking. Like a data flow operator, each object waits until it has all the inputs it expects and then "fires". When it fires it computes its outputs and sends them to its neighboring cells in the array.

In the implementation here, some liberties have been taken with the Z elements. In the program, these delay or storage elements are placed in the cells such that each cell uses the current inputs and never the previous inputs. The dotted line in Figure 2-1 shows the contents of each type of cell. Also, to avoid unnecessary complexity in this example, the extraction of the upper triangular matrix from the stacks is omitted.

In the example below, a super class Cell is defined. The attributes of Cell are inherited by each of its subclasses CenterCell, SideCell, StackCell and InputCell. It also defines the virtual procedure TryToFire which is to be found in each of the subclasses. Not all of the subclasses will make use of all the attributes of class Cell. Other than the Init attribute of Cell which sets the objects pointers to its neighbors in the array, the other attributes are defined for passing data values. When a Cell object receives a data value, it saves it, sets a flag and tries to fire. The SELECT statement prevents more of the same data item from being accepted until the object actually fires and resets the flags. If the object has all the values it requires then it does fire. When it fires, it clears all its flags and then waits for more messages.

Aside from the objects visible in Figure 2-1, an InputCell class is also defined to provide input data to the top and right sides of the array. These objects contain a stream of input data set up by another program object. The streams must contain enough "dummy" data values at the end and at the beginning to cause all the real data in the array to be pushed into the StackCell objects as the Gaussian elimination process completes.

```
CLASS GaussElim;
BEGIN

    CLASS Cell;
    BEGIN
      VIRTUAL: PROCEDURE TryToFire;
      BOOLEAN HaveLeft,HaveLower,HaveDiag;
      REAL Left,Lower,Diag;
      REF(Cell)LeftCell,RightCell,UpperCell,LowerCell,
            UpDiagCell,LoDiagCell;

      PROCEDURE Init(Lt,Rt,Up,Lo,Ud,Ld)
      REF(Cell)Lt,Rt,Up,Lo,Ud,Ld;
      BEGIN
          HaveLeft := HaveLower := HaveDiag := FALSE;
          LeftCell :- Lt;
          RightCell :- Rt;
          UpperCell :- Up;
          LowerCell :- Lo;
          UpDiagCell :- Ud;
          LoDiagCell :- Ld;
      END of Init;

      PROCEDURE TakeLeft(Val); REAL Val; SELECT (NOT HaveLeft);
      BEGIN
          Left := Val;
          HaveLeft := TRUE;
          TryToFire;
      END of TakeLeft;

      PROCEDURE TakeLower(Val); REAL Val; SELECT (NOT HaveLower);
      BEGIN
          Lower := Val;
          HaveLower := TRUE;
          TryToFire;
      END of TakeLower;

      PROCEDURE TakeDiag(Val); REAL Val; SELECT (NOT HaveDiag);
      BEGIN
          Diag := Val;
          HaveDiag := TRUE;
          TryToFire;
      END of TakeDiag;

    END of CLASS Cell;

    Cell CLASS CenterCell;
    BEGIN

      PROCEDURE TryToFire;
      IF HaveLeft AND HaveDiag AND HaveLower THEN BEGIN
          HaveLeft := HaveDiag := HaveLower := FALSE;
          RightCell.TakeLeft(Left);
```

```
        UpperCell.TakeLower(Lower);
        LoDiagCell.TakeUpper(Diag-Left*Lower);
    END of TryToFire;

END of CLASS CenterCell;

Cell CLASS SideCell;
BEGIN

    PROCEDURE TryToFire;
    IF HaveDiag AND HaveLower THEN BEGIN
        HaveDiag := HaveLower := FALSE;
        RightCell.TakeLeft(Diag/Lower);
    END of TryToFire;

END of CLASS SideCell;

Cell CLASS StackCell;
BEGIN
    REF(Stack)Stk;

    REF(StackCell) PROCEDURE SetStk(S); REF(Stack)S;
    BEGIN Stk :- S; SetStk :- THIS Cell; END;

    PROCEDURE TryToFire;
    IF HaveDiag THEN BEGIN
        HaveDiag := FALSE;
        UpperCell.TakeLower(Diag);
        Stk.Push(Diag);
    END of TryToFire;

END of CLASS StackCell;

Cell CLASS InputCell;
BEGIN
    REF(Vector)Stream;
    INTEGER I;

    PROCEDURE Go(S); REF(Vector)S; BEGIN
    BEGIN
        I := 0;
        Stream :- S;
        HaveLower := TRUE;
    END of Go;

    PROCEDURE TryToFire;
    IF HaveLower OR LowerCell==NONE THEN BEGIN
        HaveLower := FALSE;
        IF Stream.Length>I THEN BEGIN
            I := I + 1;
            LoDiagCell.TakeDiag(Stream.Element(I));
        END;
    END of TryToFire;
```

```
END of CLASS InputCell;

REF(GaussElim) PROCEDURE Init(N,M,V); INTEGER M,N; REF(Vector)V;
BEGIN
   REF(Cell) ARRAY A[0:N+1,0:M+1]
   INTEGER I,J;
   FOR I:=0 STEP 1 UNTIL N+1 DO BEGIN
        FOR I:=0 STEP 1 UNTIL M+1 DO BEGIN
           IF J=0 THEN A[I,J] :- NEW InputCell
           ELSE IF J=M+1 AND I<N+1 THEN
                A[I,J] :- NEW StackCell.SetStk(NEW Stack)
           ELSE IF I=0 THEN A[I,J] :- NEW SideCell
           ELSE IF I=N+1 THEN A[I,J] :- NEW InputCell
           ELSE IF I<N+1 THEN A[I,J] :- NEW CenterCell;
        END;
   END;
   FOR I:=0 STEP 1 UNTIL N+1 DO BEGIN
        FOR J:=0 STEP 1 UNTIL M+1 DO BEGIN
           IF J=0 AND I>0 THEN
                A[I,J].Init(NONE,NONE,NONE,A[I,J+1],NONE,A[I-1,J+1])
           ELSE IF J=M+1 AND I<N+1 THEN
                A[I,J].Init(NONE,NONE,A[I,J-1],NONE,A[I+1,J-1],NONE)
           ELSE IF I=0 THEN A[I,J]
                .Init(NONE,A[1,J],A[I,J-1],A[I,J+1],A[I+1,J-1],NONE)
           ELSE IF I=N+1 THEN
                A[I,J].Init(NONE,NONE,NONE,A[I-1,J],NONE,A[I-1,J+1])
           ELSE IF I<N+1 THEN
                A[I,J].Init(A[I-1,J],A[I+1,J],A[I,J-1],A[I,J+1],
                          A[I+1,J-1],A[I-1,J+1]);
        END;
   END;
   FOR I:=1 STEP 1 UNTIL N DO A[I,0] QUA InputCell.Go(V[I]);
   FOR J:=1 STEP 1 UNTIL M DO A[N+1,J] QUA InputCell.Go(V[J+N]);
END of Init;
END of CLASS GaussElim;
```

The Init attribute of class GaussElim constructs an array of Cell objects. It first creates the objects, putting pointers to them into an array. It then makes another pass through the array initializing the Cell objects with pointers to their neighbors. Then the InputCell objects along the upper and right sides of the array are each given a stream of input data permitting them to begin pushing the data into the array. After all the data in the streams is exhausted, the upper triangular matrix is stored in the StackCell objects along the bottom of the array.

When the array cells are initialized the flags in the InputCells are set on. This action puts the array in a state from which it will proceed. Until the matrix has filled the array, zero data values will circulate in the array. This type of operation is the same as that defined for the hardware implementation of [Johnsson81].

For this example it is evident that the concurrency and synchronization required for data flow problems is available in this programming model. The SELECT statement, used to control under what conditions attributes may be selected, is the means for synchronizing each object with the availability of its multiple inputs. Unlike the two previous example programs, the program for Gaussian elimination is not executable as a normal sequential Simula program, as the SELECT statement has no meaning in sequential environment.

2.6. Comparison with CSP

The CSP notation [Hoare78] is a means of expressing concurrent programs. This notation includes the concept of processes and messages. Explicit operators represent the sending and receiving of messages. Messages are not queued and require the sender to execute a send operation and the receiver to execute the receive operation at the same time. This action results in a strong means of synchronizing two processes.

The Bounded Buffer problem described by Hoare is shown below in CSP notation. This CSP program describes a process X which takes objects of type "portion" transmitted by "producer" and stores them in "buffer" until requested by "consumer". In CSP notation the source and destination of send and receive operations are unique processes.

```
X::
buffer:(0..9) portion;
in,out:integer; in:=0; out:=0;
comment 0≤out≤in≤out+10;
     *[in < out + 10; producer?buffer(in mod 10) → in:= in + 1
   □ out < in; consumer?more() → consumer!buffer(out mod 10);
       out := out + 1
     ]
```

If process X is considered to be an object, and producer and consumer its attributes, each with an associated message queue, we see that process X can implement a FIFO object for any number of producers and consumers. Any object that produces objects of type portion may send the portions to X using the producer attribute. As long as the buffer array is not full, these messages will be accepted by the input command and the portion stored in buffer. Objects that request portions may do so by invoking the consumer attribute. As long as the buffer is not empty, process X will answer consumer messages with the transmission of a portion.

The following code is a concurrent Simula description of a FIFO object.

```
CLASS Fifo;
BEGIN
     REF(Portion) ARRAY Buffer[0:9];
     INTEGER In,Out;

     PROCEDURE Put(P); REF(Portion)P; SELECT In < Out+10;
     BEGIN
        Buffer[Mod(In,10)] := P;
        In := In + 1;
     END of Put;

     REF(Portion) PROCEDURE Get; SELECT Out < In;
     BEGIN
        Get :- Buffer[Mod(Out,10)];
        Out := Out + 1;
     END of Get;

END of CLASS Fifo;
```

A close correspondence can be observed between the guarded input commands of the CSP representation of the FIFO and the attribute

declarations of concurrent Simula with their SELECT statements. Concurrent Simula permits the definition of objects which are a subset of CSP processes. As part of the language, concurrent Simula builds a repetitive command around a series of alternative guarded input commands to implement the selection and initiation of an object attribute. The Boolean expressions that may be associated with the SELECT statement are the guards for the attributes. Outputs from the attributes are implied by their termination rather than programmed explicitly.

CSP is a more expressive notation that, if used to describe the objects of the homogeneous machine, permits a more concise representation than concurrent Simula. In particular, CSP provides a convenient means of expressing the input of several messages concurrently. Using a CSP style notation, the CenterCell of the Gaussian elimination array would be described in the following manner.

```
CenterCell::
DiagC,LowerC,LeftC:Cell;
RightCell,UpperCell,LoDiag:Cell;
D,L,Low:Real;
    *[DiagC?Diag(D) Λ LowerC?Lower(Low) Λ LeftC?Left(L) →
      RightCell!Left(L);
      UpperCell!Lower(Low);
      LoDiag!Diag(D-L*Low);
     ]
```

A notation like CSP has the necessary expressiveness to cope with objects having a message queue per attribute. The semantics of message passing in the object oriented machine are somewhat different than those defined by Hoare but can nevertheless be adapted to CSP notation. The CSP description is more concise than the concurrent Simula description of the example.

A CSP notation could be used to program the homogeneous object-oriented machine. It has explicit provisions for concurrency and message passing. However, the explicit sending and receiving of messages put a greater burden on the programmer by requiring the synchronization and mutual exclusion needs of the program to be explicitly described, thus expanding the opportunities for error.

2.7. Support Requirements

Arbitrary pointer topology provides clear possibilities for deadlock between objects. Deadlock can but will not necessarily occur wherever the user has generated a loop in the communication structure of the program components. One method of preventing this possibility would be to restrict the topology to a non-cyclic graph such as a tree. Such restrictions appear to be so constraining as to render many of the advantages of this programming style useless.

The most immediate deadlock situation can arise when an object holds a pointer to itself. If, in the course of satisfying some message it has received, it uses that handle and sends a message to itself for which it expects a response, the object will be stuck. The new message it has sent to itself cannot be acted upon and a response sent until it has completed the execution of the current message. However, the completion of the current message is awaiting a response to the new message. At this point the object is hopelessly deadlocked. It is important to note that the mere holding of a pointer that points to the object that holds it does not inevitably result in deadlock. The code of the objects and the topology of the pointers must conspire to create the deadlock situation. The example cited is the simplest case of situations that may arise whenever a cycle appears in the

communication between objects.

For the purposes of this programming model, the object messages queues are of a fixed size. The size can be either declared as part of the object declaration or can by set to a default size by the system. If no limit is placed on the size of a queue, then incoming messages could cause a queue to grow to arbitrary size if the destination object is unable to service the messages at the rate they arrive. Such growth could cause the space allocated for message queues to become exhausted and deadlock to occur if there is no limit on the length of message queues. With a limit on message queues, objects sending messages to other objects may be suspended until there is space available in the queue.

Deadlock can also occur with bounded queues where all the queues in a cycle become full with each object attempting to transmit a message to another. This condition must be recognized and dealt with by the programmer. Bounded queues permit deadlock only among the objects which own the queues. An attempt to simulate infinite queues in a system with finite resources introduces possibility of deadlock to the processors, a situation that can not be permitted.

To prove a program to be deadlock free would require some restriction of the use of object pointers. Some types of communication structures have been proved safe and live [Owicki80], however, these structures represent a severe restriction of the structures possible in the object environment presented here. It may be possible to extend the proof techniques of [Chen82] to aid in the prevention of deadlock among the objects of a program.

The run time system which implements message passing and garbage collection has internal access to all objects. For the purposes of debug only, we can define a set of attributes that are built-in to all objects and that are recognized by the run-time system. These attributes would have to return the following information about an object.

(1) Object is Idle, Running or Waiting.

(2) If Waiting, return a reference to the object it is expecting a response from.

(3) Return the object's stack and current place of execution in its code.

(4) Return any of the object's internal data items.

(5) Return a list of messages waiting to be processed by the object.

Given these abilities, a debugger could give the user a complete picture of the state of a program. The debugger should also be able to manipulate objects in an equivalent manner to allow the user to break a deadlock or modify the state of an object.

The ability to manipulate objects via the run-time system implies that some protection is required to prevent objects from arbitrarily corrupting each other. One solution to this problem is the addition of protection or capabilities to object references [Jones73]. Various fields in the object reference would determine the legal operations that the holder of the reference could expect the referenced object to perform.

The addition of capabilities to object references would not increase the security of the system unless object references are made unforgeable. To accomplish this goal, a tagged architecture such as the SYMBOL machine [Rice71] or the Intel 432 [Kahn81] is required. A tagged architecture is a

machine that recognizes data types at the hardware or firmware level. The price for such security is usually reduced performance.

2.8. Conclusions

We have presented a model for concurrent programming based on the Simula object concept. The object is not unlike the distributed process construct of Brinch Hansen [Brinch Hansen78]. Objects communicate and synchronize by passing messages. An object holding a reference to another object is permitted to send messages to it and may optionally receive a response. The messages directed at any given object are delivered via attribute queues which order the messages and implement critical regions within the object.

The sending of a message from one object to another is initiated by invoking an object's attributes. If a reply to the message is required, sequential execution of the object's attributes proceeds while the requesting object waits. If no reply is desired, concurrent behavior is initiated. Both the requesting object and the requested object may continue execution after the fashion of processes in the Multics system [Spier69] or as would the coroutines of Concurrent Pascal [Brinch Hansen75].

The message queues are ordered and messages may be removed only after the termination of the actions associated with the previous message. This feature makes each attribute of an object a critical region so that there is mutual exclusion between those routines that have access to the internal variables of the object. Synchronization between objects is also provided by insuring that all messages preceding a message in a queue must have been acted upon before a response to the message can be sent.

Concurrently executing processes must be prevented from modifying state in an order not intended due to races between them. Critical regions, for example, are intended to give the programmer a means of preventing such behavior. The extensions to object oriented programming put all the routines that have access to common variables in mutually exclusive regions as a natural part of the language. The implementation is accomplished by the use of queues. The programmer is left with the responsibility for insuring that all sequences of messages are either legal or the programmer must control the possible sequences of the program. Synchronization with the completion of one object's task is accomplished by merely querying the object in question. The arrival of the response indicates the desired state has been reached and thus synchronizes one object with another.

The extensions and restrictions proposed for a language like Simula add little additional complexity to the syntax of the language. The effects are most felt in the more restricted scoping rules preventing global variables. Example programs show that this notation can be conveniently used to write concurrent programs. Sufficient means are available to the programmer to insure repeatable behavior in the programs.

Chapter 3

Garbage Collection

3.1. Introduction

Presented here is a scheme for the identification and elimination of inaccessible program objects in a large multi-processor environment. The problem of garbage collection has been an interesting problem for many years among the implementors of various languages such as Algol 68 [Van Wijngaarden69], Simula 67 [Birtwhistle73] and LISP [McCarthy60] which provide for dynamic allocation of data structures. Garbage collection has been part of operating systems to a lesser degree for sometime but took on a new importance with the implementation of Hydra on C.mmp [Wulf72,80]. Here, the operating system is distributed over a number of processors and therefore, collection of garbage must take place across a number of distinct address spaces concurrently with the operation of a number of processors [Almes80].

Various implementations of LISP have dealt with the problem of garbage collection. Early work [McCarthy60,Collins60] provided garbage collection for LISP on a single processor. More recently, considerable effort has been given to the use of multiple processors to execute LISP with at least one of them responsible for garbage collection [Steele75,Deutsch76,Wadler76]. The algorithm proposed by Dijkstra, et al [Dijkstra78] has been proved correct [Gries77]. In all cases, these algorithms are presumed by their authors to

operate in a system where every processor has equal or near equal access to a single address space. Moreover, the problem of garbage collection in pure LISP is somewhat more restricted than the more general case of Simula [Arnborg72], insofar as LISP objects are of fixed size and LISP data structures may be of a restricted topology.

Garbage objects can be identified in systems by reference counting [Collins60]. This technique can be applied to most systems, even those with more than one processor. However, reference counting suffers from two problems for which no acceptable solutions come to mind. First, self-referential data structures or data structures with cyclic graphs can not be identified as garbage by this method without the addition of multiple levels of reference counts and a grouping concept as in [Bobrow80]. For some environments, those which restrict the user to tree-like structures, this problem may be tolerable but in a more general system it is not. Reference counting also involves a very large computational overhead to keep the reference counts up to date. Since each object's reference counter must be modified whenever and wherever a pointer to that object is copied or overwritten, many simple operations become complex. In a multiprocessor system, this overhead is manifested either by a high communication traffic or by a large number of memory accesses used to update reference counts as pointers are manipulated.

An object oriented system provides some simplifications not possible in the LISP systems but also introduces new complexities. If each processor is to control the access and function of its own set of objects, then the need for notions of mutual exclusion, critical regions and indivisibility in the operations of the processor is eliminated since it is the only entity in contact

with its objects. However, the interactions among the processors of the system raise new problems, such as their synchronization if they are to perform such tasks as garbage collection.

3.2. The Object-Oriented Environment

Briefly, the environment consists of a large number of objects with a structure of pointers between them of an arbitrary topology. The objects are distributed over a number of processors and those objects that are executing may change the topology of the pointers. Pointers are passed from object to object in messages and objects may also be moved from one processor to another. The object of garbage collection is to identify and eliminate those objects which are inaccessible in the system and are idle. The elimination of the garbage objects allows the resources occupied by them to be allocated to new objects as they are created.

3.3. A Description of the Algorithm

Every processor node which executes, stores or otherwise manipulates objects must run a task in the background which is part of the overall garbage collection process. These tasks, each in one of the processors in the network, communicate with a central process which maintains overall control of each phase of the garbage collection. The communication between the central process and the various tasks in the processors is of a very low bandwidth and serves only to synchronize the other tasks in a very coarse way with respect to the stages of the garbage collection. The central process may be implemented as a separate processor with dedicated communication facilities to connect it to the other processors, or it may be merely another background task executing in any processor and communicating via the

same medium as the objects in the system.

There are three phases to the collection process. The first is to unmark all objects in the system. The second is to mark those objects which are not garbage. And third, all unmarked objects are collected and the resources they occupy are made available for allocation to new objects.

Some definitions are needed:

D1: A root object is one that is either executable or waiting for a response to a message. An idle object, one that is waiting for a message, can become executable if it receives a message.

D2: A propagation path consists of a set of pointers from a root object to an idle object. To send a message to an idle object, a propagation path must exist.

D3: A garbage object is one for which no propagation path exists.

The following conditions must also apply:

(1) Each object must have an attribute of MARKED. This attribute is TRUE after the garbage collection task to which this object resides has determined that this object is not garbage.

(2) Each object has an attribute of RECEIVEDMARK. This attribute is TRUE if a processor other than the one in which the object resides has determined that the object is not garbage and has sent a message to this object's processor indicating this condition. A set RECEIVEDMARK is essentially a request from one processor to another that a particular object be marked.

(3) Each reference variable must have an attribute of MARKED. If a reference variable is copied or sent to another object in a message, this

attribute is preserved in the new copy or in the message. If this attribute is TRUE, then the object that it refers to may be considered MARKED.

(4) The communication facilities must not allow messages to be hidden from all processors at any time. If messages in transit are inaccessible to processors, then a copy of the message must be kept by the sender until the message is known to have arrived at its destination. It is required that every message in the system be accessible to at least one processor at all times.

Qualitatively, the algorithm operates in the following manner. All of the garbage collection tasks are told to unmark their objects and reference variables. When this operation is completed, all the processors are told to begin marking non-garbage objects. At first, this operation consists of scanning all the objects in each processor and marking the ones that are executable and recursively marking all the objects and reference variables that can be reached by following pointers from the executable objects. If the processor determines that an object, which resides in another processor, is to be marked a message is sent to that object, wherever it does reside, to cause it to be marked by its processor.

As long as a processor is in the mark phase, it must process incoming messages in a different manner than usual. It must mark the recipient of the message (if it has not already been marked) and it must mark any objects referred to by reference variables in the message in cases where the reference variables are not marked. Thus, as the processors enter the mark phase, waves of set mark attributes emanate from executable objects and

from objects that are in involved in communication with each other. It is assumed that the participants in an exchange of messages, and objects referred to in messages, are not garbage since they are obviously in use. This use of the normal communication between objects as part of the marking process speeds up the rate at which garbage can be collected but does not add to the message traffic. In effect, the object communication performs a double duty during the mark phase. It accomplishes the function programmed in the objects as well as identifies the objects involved as non-garbage.

When the marking of objects has finished, the remaining unmarked objects are collected as garbage. The resources belonging to these objects, their name and disk space, are released for use by new objects. The cycle is then repeated by again clearing all the mark attributes.

The processor controlling the phases of garbage collection executes the following task. The processor executing this task may be multiplexed among other tasks as well, or it may be a dedicated processor. The algorithm is described in a Simula-like syntax. Procedures such as "SendMessage" and "SendMessageToAllProcessors" are not shown in detail since they depend on the particular hardware and software communication facilities available. It is hoped that the function of the undefined procedures is self-evident.

```
WHILE TRUE DO BEGIN

    PROCEDURE WaitForAllDone;
    BEGIN
       BOOLEAN Done;
       Done := FALSE;
       WHILE NOT Done DO
       BEGIN
            SendMessageToAllProcessors("StartInterval");
            WaitUntilAllAcknowledge;
            SendMessageToAllProcessors("EndInterval");
            WaitUntilAllAcknowledge;
            Done := ANDofAllDoneFlags;
       END of While;
    END of PROCEDURE WaitForAllDone;

    SendMessageToAllProcessors("ClearAllMarks");
    WaitForAllDone;
    SendMessageToAllProcessors("EndClearAllMarks");
    WaitForAllDone;
    IF ThereIsARootObject THEN BEGIN
       REF(Processor)Root;
       Root :- ProcessorWithRootObject;
       Root.SendMessage("MarkRootObject");
    END of IF;
    SendMessageToAllProcessors("MarkExecutableObjects");
    WaitForAllDone;
    SendMessageToAllProcessors("CollectUnmarkedObjects");
    WaitForAllDone;
    SendMessageToAllProcessors("EndCollectUnmarkedObjects");
    SendMessageToAllProcessors("EndMarkingExecutableObjects");
    WaitForAllDone;
END of Garbage Collector Control Loop;
```

The loop above contains no "critical regions" and none of its operations must be "indivisible". If this task shares a processor with other tasks, the processor may be removed from this task at any point in the loop. The only effect such multiplexing may have is to reduce the rate at which garbage collection proceeds by a very small amount, providing this task receives even minimal service from the processor. The exclusive access given a processor to the objects contained in its memory simplifies the interactions between processors. The synchronization, mutual exclusion and other conditions that must be met are embedded in the sequence of message passing. The

indivisible operations that must exist in such a system are those of message transmission and reception.

The messages sent to all the processors could be broadcast, if the connection medium permits it. The "WaitUntilAllAcknowledge" procedure must hold further execution until it is known that every processor has received the previous message. This operation is the primary means by which the processors and the controlling garbage collector task are synchronized. This synchronization is of a very weak nature. The acknowledgement of the processors could be detected by waiting until all the processors pulling down an open-collector TTL signal have released it, or it could be detected by the receipt of an acknowledging message from each processor, depending on the communication facilities present. Some of the message sequences in the above loop could be concatenated into single messages but have been separated for clarity.

The "ANDofAllDoneFlags" is a hypothetical procedure which returns TRUE if the "DoneFlag" (described below) of every processor is TRUE. This function could be performed by querying each processor with an exchange of messages or with hardware, such as an open-collector signal wired to all the processors. The "DoneFlag" is defined to be valid at the time a processor does the "AcknowledgeMessage" operation and until it receives its next "StartInterval" message.

The determination of when all the marks in the system are clear or when all the garbage objects have been collected or, most importantly, when all the non-garbage objects have been marked is the mechanism that permits this algorithm to work. The most difficult question is how to determine when the marking of non-garbage objects is complete and to be assured that no

more objects can be or will be marked. The collection phase cannot be initiated until the marking is finished.

The "StartInterval" and "EndInterval" messages from the controlling task delimit a span of time in each individual processor. The sequencing of the controlling task insures, despite any skew in the arrival of the messages at the processors, that a sub-interval of all the spans of time is common to all the processors in the system.

It can be said that if, during some interval of time, not one of the processors in the system marked any objects nor had any objects that were waiting to be marked, no further marking can occur in the system. When an processor receives a "StartInterval" message during a mark phase (but not while in a collection phase) it scans all its objects for any that should be marked but are not. If any objects are marked by the processor, its "DoneFlag" will subsequently exhibit FALSE. During the interval the processor may mark objects and will again record the fact if any are marked. When a "EndInterval" message is received and no objects have been marked since the interval began, the processor will again scan its objects and record any that are marked. At the end of the interval the "DoneFlag" is displayed indicating, if TRUE, that no marking was done or could have been done during the interval in that processor.

If all the processors display a TRUE "DoneFlag" at the end of an interval, then there were no objects marked in the entire system during that portion of the interval shared by all the processors. It follows that if, over the entire system, no objects were marked and no objects were waiting to be marked, then the mark phase has finished. An object must be marked to cause other objects to be marked. Therefore, if there are none to be marked and no

marking has been done, there can be no further marking.

Several aspects of the sequence of messages initiated by the controlling task should be noted. The collection phase has been made a part of the marking phase. This relationship insures that any new objects created before and during the collection phase are created marked and are prevented from being collected as garbage. Otherwise, new objects could be created with a FALSE mark bit after the marking is completed but before collection, causing any such objects to be regarded as garbage. The overlapping of the mark phase with the collection phase prevents this situation.

In addition, there is what might be regarded as a spurious "WaitForAllDone" procedure inserted between the end of the clear phase and the beginning of the mark phase. This invocation serves only to insure that all of the processors have stopped clearing prior to beginning to mark. If the situation arose where some processors were already into the next mark phase before others had recognized the end of the previous clear phase, not only would confusion result in the state of various mark bits, but neither set of processors could complete their respective phases since messages would continue to arrive with marks in an unexpected state.

A detailed description of the functions that must be performed by each processor as part of garbage collection is below. This description is shown as a message dispatch routine that intercepts and disposes of all the incoming messages of a processor. In an actual system, the mechanism associated with receiving messages from the garbage collection controller may be quite separate from the facilities used to process messages from other processors.

The procedure below would be entered when a complete message is available to the processor. Upon returning from the procedure the processor's scheduler would select other tasks for execution. In the form shown here, this procedure cannot be interrupted for the execution of objects, but may be interrupted for other tasks.

```
PROCEDURE DispatchMsg(Message); REF(Msg)Message;
BEGIN
    BOOLEAN Clearing,Marking,Collecting,DoneFlag;

    PROCEDURE MarkObject(abc); REF(Object)abc;
    IF NOT abc.Marked THEN BEGIN
       REF(ReferenceVariable)RefVar;
       abc.Marked := TRUE;
       abc.ReceivedMark := FALSE;
       DoneFlag := FALSE;
       FOR RefVar :- abc.EachRefVarInThisObject DO BEGIN
            IF NOT RefVar.Marked THEN BEGIN
               RefVar.Marked := TRUE;
               IF RefVar.Object.InThisProcessor THEN
                    MarkObject(RefVar.Object)
               ELSE RefVar.Object.SendMessage("TurnOnReceivedMark");
            END;
       END of FOR Loop;
    END of PROCEDURE MarkObject;

    PROCEDURE DoFunction;
    BEGIN
       REF(Object)Obj;
       IF Clearing THEN BEGIN
            FOR Obj :- EachObjectInThisProcessor DO BEGIN
               IF NOT Obj.AllClear THEN BEGIN
                    ClearAllMarkBits(Obj);
                    DoneFlag := FALSE;
               END;
            END of FOR;
       END ELSE
       IF Marking AND NOT Collecting THEN BEGIN
            FOR Obj :- EachObjectInThisProcessor DO
               IF Obj.Executable OR Obj.ReceivedMark THEN MarkObject(Obj);
       END ELSE
       IF Collecting THEN BEGIN
            FOR Obj :- EachObjectInThisProcessor DO BEGIN
               IF NOT Obj.Marked THEN BEGIN
                    RecoverGarbageObject(Obj);
                    DoneFlag := FALSE;
               END;
            END of FOR;
       END of IF;
    END of PROCEDURE DoFunction;

    IF Message.Destination=GarbageCollector THEN BEGIN
       IF Message.Txt="ClearAllMarks" THEN Clearing:=TRUE
       ELSE
       IF Message.Txt="EndClearAllMarks" THEN Clearing:=FALSE
       ELSE
       IF Message.Txt="MarkExecutableObjects" THEN Marking:=TRUE
       ELSE
       IF Message.Txt="CollectUnmarkedObjects" THEN Collecting:=TRUE
```

```
        ELSE
        IF Message.Txt="EndCollectUnmarkedObjects" THEN Collecting:=FALSE
        ELSE
        IF Message.Txt="EndMarkingExecutableObjects" THEN Marking:=FALSE
        ELSE
        IF Message.Txt="MarkRootObject" THEN MarkObject(RootObject)
        ELSE
        IF Message.Txt="StartInterval" THEN BEGIN
            AcknowledgeMessage;
            DoneFlag := TRUE;
            DoFunction;
        END ELSE
        IF Message.Txt="EndInterval" THEN BEGIN
            IF DoneFlag THEN DoFunction;
            AcknowledgeMessage;
        END;
    END ELSE
    IF Message.IsObjectTransfer THEN BEGIN
        REF(Object)Obj;
        Obj :- Message.AsObject;
        IF Obj.Marked AND NOT Marking THEN BEGIN
            Obj.Marked := FALSE;
            Obj.ReceivedMark := TRUE;
        END;
        PutObjectInProcessor(Obj);
    END ELSE BEGIN
        REF(Object)Obj;
        Obj :- Message.Destination;
        IF Message.Txt="TurnOnReceivedMark" THEN BEGIN
            IF Marking THEN MarkObject(Obj)
            ELSE Obj.ReceivedMark := TRUE;
        END ELSE
        IF Marking THEN BEGIN
            REF(ReferenceVariable)RefVar;
            MarkObject(Obj);
            FOR RefVar :- Message.EachRefVar DO BEGIN
                IF NOT RefVar.Marked THEN BEGIN
                    RefVar.Marked := TRUE;
                    IF RefVar.Object.InThisProcessor THEN
                        MarkObject(RefVar.Object)
                    ELSE RefVar.Object.SendMessage("TurnOnReceivedMark");
                END;
            END of FOR Loop;
            GiveMessageToObject(Message);
        END ELSE GiveMessageToObject(Message);
    END of IF;
END of Message Dispatcher;
```

One important part of the algorithm cannot be represented as part of a message dispatch routine. This part of the algorithm must be invoked

whenever a new object is to be created in a processor. It can be stated simply as follows:

```
IF Marking THEN BEGIN
    REF(Object)Obj;
    Obj :- TheNewlyCreatedObject;
    Obj.Marked := TRUE;
END of IF;
```

This provision exists to insure that all objects created in a processor while that processor is in a mark phase are created MARKED to prevent their premature collection in the next phase.

Messages representing objects that have been moved from one processor to another are accounted for in the message dispatch procedure. The only requirement placed by this algorithm on such messages is that if a marked object arrives at a processor that is not yet in the mark phase, that object becomes unmarked and acquires the attribute of RECEIVEDMARK before becoming a *bona fide* resident of the processor. When the receiving processor enters the mark phase, it will note the attribute of RECEIVEDMARK in the object and will mark it on the first pass.

In the "MarkObject" procedure, the attribute "EachRefVarInThisObject" is taken to return each reference variable associated with the object in question. Reference variables contained in unprocessed messages, or contained in an internal stack must be included as well as those that are part of the visible state of the object.

The generation of garbage by the system continues without regard for the phases of garbage collection. At any time, reference variables may be overwritten with other reference variables. When all the reference variables pointing a set of non-executable objects are destroyed, the objects become garbage. This process occurs during the mark and collection phases and at

all other times as well. Objects that have been marked and subsequently become garbage will not be collected in the next collection phase. However, it is guaranteed that the next time around through the mark phase, they will not be marked and hence will be collected in the next cycle.

One refinement of the above algorithm would eliminate the clear phase. After all the unmarked objects have been collected in the collection phase, the remaining objects and their reference variables must all be marked. Thus, only the sense of the mark bits needs to be changed to consider the system cleared. A mark pass must set all the marks in the system to the same value. In the routines above the value is TRUE (presumably a one). If, instead of sending the "ClearAllMarks" message, a message indicating "InvertMarkSense" was sent, then on the next pass, a mark with a one in it would be considered unmarked rather than marked. After that pass the sense would again be inverted and so forth after each pass. This refinement has not been shown in the algorithm to preserve its readability. If this technique were adopted, a substantial fraction of the garbage collection overhead would be eliminated.

3.4. Proof

An informal proof is presented here that the garbage collection algorithm operates correctly in the environment outlined.

By definitions D2 and D3, no root object can send a message to a garbage object. Therefore, garbage objects can never become executable and will remain idle. By D1 and D2, any idle objects that become executable indicate the existence of a propagation path and cannot have been garbage by D3. This proves that the set of root objects at any point in time is a sufficient set from which to begin marking.

The algorithm, as presented here, does its marking as part of a message dispatch service. This implementation makes the entire recursive marking of an object by the "MarkObject" procedure indivisible. This rather strong restriction may not be necessary but enables other properties of the algorithm to be studied. This proof assumes that the marking of each individual object, together with the object's reference variables is one indivisible operation.

The following invariant relations must hold:

P1: If an object is marked, all of the pointers contained in it are also marked.

The marking procedure marks an object and all its pointers in one operation. All the pointers will remain marked if no unmarked pointers are sent to the object in a message. All messages to an object are scanned for unmarked pointers before being given to the object, assuring the P1 remains true if the object remains in one processor.

The only case whereby an object may be unmarked is when it is moved to a processor that is not in the mark phase. In this case, the object is unmarked and its "ReceivedMark" flag is set to insure the processor marks it when marking is eventually begun in that processor. Since messages passed in a processor that is not in the mark phase may contain unmarked pointers, a marked object moved into that processor is unmarked to preserve the truth of P1.

P2: If a pointer is marked, then the object it points to must be either marked or have been sent a message causing its "ReceivedMark" flag to be set.

The "MarkObject" procedure and the code that scans messages for unmarked pointers are the only points at which a pointer in a reference variable is marked. Since at both places, the object referred to is either marked or sent the "ReceivedMark" message, P2 is maintained.

P3: For each unmarked non-garbage object, there exists a propagation path.

At the beginning of marking, a propagation path exists to all non-garbage objects by the inverse of D3. The marking of the data structure cannot modify the data structure and cannot, therefore, break a propagation path. The only modification that the running objects can make on the data structure is to redirect a reference variable from one non-garbage object to another non-garbage object (newly created objects are non-garbage and are created marked). The object pointed to by the modified reference variable clearly has a propagation path, since by D1, the object pointing to it is a root object. The original object pointed to may become garbage following the modification and would not then violate P3. If two or more propagation paths existed for the original object, then P3 is preserved since at least one propagation path will remain.

For the algorithm to perform correctly, the following correctness criteria must be met.

CC1: All garbage objects present at the start of marking will never be marked.

CC2: At the completion of marking, no non-garbage object remains unmarked.

If the root objects are marked, as shown in the "DoFunction" procedure, then by P3, there will be a path from a marked object to all unmarked non-garbage objects. P2 and P3 insure that once marked, objects and pointers

will remain marked. It remains to prove that marking will complete with CC1 and CC2 true.

By D3, there can be no propagation path to an object that is garbage from the beginning of marking. To become marked, the object must be sent a message by a root object or it must be referred to in a message from a root object. If no propagation path exists from a root to a garbage object, it can never be sent a message, preventing the "MarkObject" procedure from being invoked on it. This assures that CC1 can never be violated.

To complete marking, a stable, detectable state must be reached. This state must satisfy CC2. If all the processors eventually begin the marking of objects, then all root objects will be marked. In the process of marking objects, "ReceivedMark" messages are sent to objects referred to in other processors. These messages can be produced only along existing propagation paths, since marking begins with the root objects. These messages are only produced when an object is marked. If CC1 is satisfied, then when all non-garbage objects are marked, no such additional message can be produced. If all objects are marked, then by P1 and P2, all pointers of non-garbage objects are marked. Since only non-garbage objects may become root objects by D1 and D2, all pointers contained in messages produced by root objects must be marked. In this state, where all executing (root) objects are marked, their messages contain only marked pointers and where no objects have a "ReceivedMark" flag set, is stable because none of the conditions that would cause "MarkObject" to be invoked exist.

This state satisfies CC2 because propagation paths remain to be followed only as long as there are outstanding "ReceivedMark" flags. The "MarkObject" procedure would complete the marking of all the objects with

propagation paths to a single root if all the objects were in the same processor. Where the path leads out of the processor the "ReceivedMark" message is used to cause another processor to continue the marking of a path. As long as propagation paths remain to be marked, there must be outstanding "ReceivedMark" flags. When no such flag is set in the system, CC2 is met and marking is complete.

To detect completion, the "StartInterval" and "EndInterval" messages are used to delimit a period of time that is shared by all the processors in the system. At the beginning of the interval, the processors scan their objects for any that require marking, notably ones with the "ReceivedMark" flag set. If any are marked, the fact is recorded. The processors also detect whether any objects were marked or required marking during and at the end of the interval. If, over all the processors, no objects were marked and hence none had a true "ReceivedMark" flag, the marking has completed.

3.5. Simulation Results

To support the contention that the garbage collection algorithm performs as described above, a discrete simulation of its components was written and run giving every indication that it is a viable technique. The simulation was implemented in Simula using the Demos simulation package [Birtwhistle79]. A stochastic model of the executing objects was using to represent a system of running objects. The objects were given the necessary attributes and placed among a set of simulated processors each containing the garbage collection routines defined above.

The model used to represent the executing objects was a set of probabilities picked to insure that all the pathological cases of garbage collection were well exercised. In the absence of any experience or data

available for concurrent programs executing on a collection of processors, the numeric values were picked to be both acceptable within the scope of experience on uni-processor systems and to be a true test of the algorithm. The resulting simulation shows that for the situations encountered using the model, the algorithm performed as expected. Some statistics were derived from the simulation but these are more a description of the simulated environment than a prediction of efficiency or performance.

The following is a detailed description of the model used for simulation.

(1) The basic time-slice interval of a processor was an average of .0167 seconds with a standard deviation of .008 seconds. The time-slices of each processor varied about the mean with a normal distribution.

(2) The probability that a given time-slice was used by a processor to service its garbage collection task was 0.50

(3) The size of an object, in terms of the number of reference variables it held was a normal distribution with a mean of 12 and a standard deviation of 10. Once created, the size of the object remains fixed.

(4) If a time-slice was used by a processor to execute objects, the number of objects "touched" in that time was a uniform distribution from 1 to 5 (inclusive).

(5) If an object was touched, the probability that it completed its execution, becoming idle to await another message, within that time-slice was 0.12

(6) The probability that it would cause a new object to be created was 0.10

(7) The probability that it would be moved to another processor if touched was 0.20

(8) The probability that it would communicate with the objects for which it had reference variables was 0.30

(9) If it communicated, the probability that any particular reference variable contained in the object or in any of its "sub" objects was transferred in a message was 0.15

Given the practical limitations of address space and processor bandwidth, no more than 36 processors could be simulated with 1800 objects between them. The machine used for the simulation was a DECsystem-2060 running Simula version 5. The simulation that produced the results below consumed approximately 10 hours of CPU time. The processors are defined to have a capacity of one third more objects than the total number of objects divided by the number of processors, giving an average utilization of 75% among the processors. No attempt was made in the simulation to provide or maintain locality between the objects. Here again, it was thought that uniform communication between objects and hence between the processors was a more rigorous test than one with some presumed degree of locality or a presumed topology.

The simulator does not presume the existence of a "root" object. After the initial set of objects is created in the simulated system, a random set amounting to 40% of the total set of objects is set to be "executable". The simulation of the system proceeds with these objects until an equilibrium is reached with some varying percentage of 1800 objects active at any given time based solely on the simulated communication between the objects.

A check is built into the simulator to verify that the garbage collection works. Prior to each mark phase, the simulated system is stopped and a

conventional garbage collection is performed to construct a list of objects known to be garbage at that instant. The objects in the list are left in place in the system and merely noted for subsequent reference. After the collection pass, the system is again stopped and the objects collected by the algorithm are compared with those noted in the list. Every object in the list must have been collected. If it was not, an error is generated since the algorithm would have failed to collect objects it should have collected. Failing to collect a garbage object would eventually cause a system to fail as the uncollected objects accumulate until they occupy all the resources of the system.

A second check makes certain that after each collection pass there are no references in the remaining objects to any of those previously collected. Again, if any object in the system is found to contain a reference to an object that the garbage collector has removed, an error is reported. Such a failure would indicate that the algorithm had falsely collected an object that was not garbage. Such an action would cause a fatal error in any real system.

At no time during the execution of the final simulator were any such errors reported. While this fact is not a rigorous proof that the algorithm is error free, it inspires a high degree of confidence that it does perform as expected. The simulated system is known to produce all the pathological pointer structures that might be expected to trouble the algorithm. Also, the built-in skew between the processors in the rate at which they poll for messages from the the controlling garbage collector task insures that synchronization problems, if any, arising from the differing states of the processors would be detected. Interprocessor interactions, such as object transfers and the transfer of reference variables in messages, are amplified

in the simulator to aggravate any possible weaknesses in the algorithm. No weaknesses have been detected by simulation.

Table 3-1

Statistical Data Taken from Simulation

Item	Observations	Average	Std.Dev.	Minimum	Maximum
Number of Objects in Existence	141754	1613.631	271.287	1031.000	1800.000
Number of Executable Objects	226594	703.869	54.122	504.000	845.000
Number of Mark Repetitions per Cycle	114	4.044	0.245	3.000	5.000
Time Required per Cycle	114	2.607	0.163	2.255	3.023
Lifetime of Objects	70877	6.816	8.088	1.205	199.451
Number of Objects Collected per Cycle	114	621.728	50.368	504.000	769.000

Table 3-2

Histogram of Number of Objects Collected Each Cycle

	Objects	Cycles	Freq	Cum	
1	0	0	0.00	0.00	I
2	75	0	0.00	0.00	I
3	150	0	0.00	0.00	I
4	225	0	0.00	0.00	I
5	300	0	0.00	0.00	I
6	375	0	0.00	0.00	I
7	450	1	0.01	0.88	I*
8	525	41	0.36	36.84	I**********
9	600	53	0.46	83.33	I**********
10	675	17	0.15	98.25	I**********
11	>750	2	0.02	100.00	I*

Table 3-3

Histogram of Object Lifetimes

	Age	Objects	Freq	Cum	
1	0	21028	0.30	29.67	I******************************
2	2	3967	0.06	35.27	I******
3	4	18995	0.27	62.07	I**************************
4	6	10811	0.15	77.32	I***************
5	8	5841	0.08	85.56	I********
6	10	2392	0.03	88.93	I***
7	12	1251	0.02	90.70	I**
8	14	1717	0.02	93.12	I**
9	16	1164	0.02	94.76	I**
10	18	781	0.01	95.87	I*
11	20	345	0.00	96.35	I.
12	22	327	0.00	96.81	I.
13	24	346	0.00	97.30	I.
14	26	293	0.00	97.72	I.
15	28	210	0.00	98.01	I.
16	30	126	0.00	98.19	I.
17	32	154	0.00	98.41	I.
18	34	184	0.00	98.67	I.
19	36	119	0.00	98.83	I.
20	38	108	0.00	98.99	I.
21	40	61	0.00	99.07	I.
22	42	85	0.00	99.19	I.
23	44	61	0.00	99.28	I.
24	46	70	0.00	99.38	I.
25	>48	441	0.01	100.00	I*

The data tabulated above show the characteristics of the running system derived from the model chosen for simulation. They also show how the garbage collection algorithm performs in this system. Of note is the maximum of 5 iterations required to mark all non-garbage objects. It is also worth noting, that in this system, approximately half of the 1600 objects in existence are executable at any given time. The "notch" in the lifetime histogram at 2 seconds is due to the cycle time of the collection process. With a cycle time of 2.6 seconds, any objects created during the mark phase

of a cycle must wait until the next full cycle to be collected if they are made garbage. This effect skews the graph of what would otherwise be a Poisson distribution. Other interesting aspects are: An object has a 90% probability of being collected as garbage within 4 cycles of the garbage collector. Typically, one third to one half the existing objects are collected each cycle. These figures indicate a rapid turnover in the objects and thus a rapidly changing data structure.

3.6. Performance Analysis

The simulation, despite the care with which the model parameters were chosen, cannot give more than clues to how the algorithm might perform in a real system. The characteristics of an actual set of objects in a real system, communicating with each other in some topology, executing and manipulating pointers in some manner and migrating between processors, is unknowable *a priori* and depends as much on the application of the system as on the system itself. However, this garbage collection algorithm, while it is intended to operate in such an environment, can be compared with conventional garbage collectors on uni-processor systems. In addition, it is important that the algorithm's performance scale, as well as the number of processors in the system increases.

The conventional garbage collection program running on a single processor system must make at least two passes across the data structure it manipulates. It must first follow every path from known non-garbage objects to mark every object that can be reached. It must then make a pass sequentially through the objects linking the garbage objects together or compacting the non-garbage objects into a contiguous area of memory. This second pass has a complexity of $O(N)$ where N is either the number of

garbage objects or the number of non-garbage objects depending on whether the linking or compaction is to be done. If N is defined to be the total number of objects, then $O(N)$ is an upper bound for the complexity of the second pass. In the first pass, every pointer must be followed to the object it points to, and if the object has not already been visited, then it is marked and all of that object's pointers must be visited. Thus the complexity of this pass is $O(N_{ob}+N_{pt})$ where N_{ob} is the number of non-garbage objects and N_{pt} is the number of pointers contained in those objects. If the worst case is assumed where none of the objects are garbage, the complexity for both passes together is $O(2N+N_{pt})$. Further, if the average number of pointers contained in an object is M then we have $O(N(2+M))$. It should be noted that both N and M are bounded by the address space of the machine and $M>1$.

For an individual processor in a multi-processor machine using the algorithm presented here, the same observations can be made concerning each pass of the combined mark and collect phases. If the data structure being collected were contained wholly within that processor the complexity of one pass would be $O(N(2+M))$ just as for the conventional garbage collector. However, the messages that result in the marking of objects in other processors and the migration of objects between processors cloud the issue. Owing to the method by which the completion of a phase is determined, both the mark pass and the collection pass are run a minimum of twice over the objects in the processor. It should be evident that the second and any subsequent scans by these routines will require less computing since most, if not all, the objects will have been touched on the first scan. If we assume the worst, this doubling of the scans increases the complexity to $O(2N(2+M))$.

In general, the passing of pointers and objects between processors and the tracing of pointer paths between processors will increase the number of scans (the number of times the loop in "WaitForAllDone" is executed) beyond the minimum of two. The volume, pointer content, locality and speed of the message traffic in the system all affect this number. Both the applications being run on the system and the hardware communication facilities will determine how long unmarked or uncollected objects can exist in the system once the mark phase has begun and thus how many scans (of "DoFunction") will be required to catch all the moving objects and reference variables. Simulation of the system where communication is very fast, locality is non-existent and pointer content of messages is high shows that no more than 5 scans are needed and typically 4 are sufficient in a system of 36 processors. For now, a number can be defined which is the average number of such scans required for each complete cycle of the garbage collection task. If this number is represented by X then the complexity of one cycle in one processor becomes $O(XN(2+M))$, where N is the number of objects held by the processor and M is the average number of reference variables in an object.

We can now compare the complexity of collecting garbage in a system with one processor versus a system with many where this algorithm is used and where the total number of objects is the same. The complexity of one garbage collection pass in a single processor system remains $O(N(2+M))$. However, the same number of objects N distributed over P processors gives a complexity of $O(\frac{XN(2+M)}{P})$. Therefore, whenever the ratio $\frac{X}{P}<1$ or when $P>X$ the multi-processor system exhibits less overhead in garbage collection than its uni-processor counterpart, all other factors being equal. If 4 is used

for X we find that 4 or more processors running the garbage collection algorithm presented here perform better than a single processor. Real values for X will have to await the construction of such machines and the accumulation of experience with their use.

Aside from the overhead incurred by garbage collection, there is a second important measure of the performance of an "on-the-fly" collection algorithm. In a conventional dynamic programming environment with a single processor, the sequential garbage collector is invoked when available memory for new objects gets low or becomes exhausted. In this type of system, the rate at which garbage is collected is made equal to the rate at which it is generated on a short term basis because the garbage collection occurs on demand. The "on-the-fly" algorithm presented here cannot be invoked on demand, but instead proceeds to completion at its own rate and then starts over. On a long term basis, it must also collect garbage at the same rate it is generated. In the short term, if system resources get low, the creation of new objects will have to wait for the completion of the current collection cycle when the resources held by garbage objects are made available.

The number of cycles the collection algorithm can perform per unit time will determine the performance of systems that run near the limit of their resources. When the resources get very low, objects which try to create new objects will be held from executing and the processors may spend inordinate effort attempting to move its objects to other processors. Of course, as more and more objects are suspended, additional processor bandwidth is available to perform garbage collection functions causing the current cycle to complete sooner. Compared to the conventional garbage

collector, the main factor in determining the speed with which cycles are completed is the factor X defined above. If X is large enough to permit short term depletion of resources in the system, then time devoted to object execution will decrease and that devoted to garbage collection tasks will increase in inverse proportion to the available resources. Providing a thrashing condition can be avoided with all the processors attempting to foist excess objects off on each other at the same time, this effect provides negative feedback to balance the effort used to garbage collect versus that used to execute objects.

3.7. Implementation Considerations

At a minimum, the integration of this garbage collection algorithm on a set of processors does not require any additional hardware beyond that which exists for the normal communication between objects. However, in some situations, using the existing facilities may not be desirable. Two capabilities make the communication between the controlling task and all the processors much more efficient and convenient. These are a broadcast capability and a wired-AND capability.

As can be seen from the description of the algorithm, the "SendMessageToAllProcessors" operation is an important one to controlling the phases of the collection process. It would be undesirable if this operation had to be implemented as the sending of an individual message to each processor in turn. If the number of processors is large, the communication bandwidth used in doing so may be unacceptable. Also, the skew introduced by the widely varied times at which the processors receive the messages will increase the overhead (i.e. the factor X will increase) and slow the rate of garbage collection. Clearly, for all but small systems, those with fewer than

about 20 processors, a broadcast capability is necessary.

The wired-AND capability is helpful in performing the "ANDofAllDoneFlags" function in the controlling task. Without the hardware to assist in this function, each processor will be required to respond with a message containing the state of its "DoneFlag". Again, the time and bandwidth used in sending such messages will have a detrimental effect on the performance of the system. The broadcast capability, if it exists, would be of no help since each processor has an individual "DoneFlag". However, since the controlling task is only interested in whether or not all of the "DoneFlags" are TRUE, a single wire using either an open-collector or open-emitter technology could be used to perform the logical AND on the wire. The "WaitUntilAllAcknowledge" function could be implemented in the same manner. A set of such signals, connected to every processor, would also make the acknowledgement of garbage collector messages by the processors much more convenient and efficient.

If a sufficient set of signals is connected to all the processors and to the controlling task, the controlling task can be reduced to a small finite state machine. A simple analysis shows that the necessary functions could be provided with less that 8 separate signals and possibly with as few as 4.

3.8. Scaling of the Algorithm

Of key importance is the ability of the algorithm to scale as the number of processors and objects increase. Specifically, is the increased computing bandwidth obtained by adding processors to the system still available as the garbage collection operates over a larger number of processors and objects? It is evident from the description of the algorithm that global communication is required between the controlling task and every other processor. In many

systems, such global communication prevents the systems from growing effectively beyond some limit imposed by the cost or the delay introduced by such communication facilities.

In the case where no broadcast facilities are present or where there is no hardware support for the "ANDofAllDoneFlags" function, there is clearly a delay in communication that grows linearly with the number of processors in the system. For small systems this delay may be tolerable but in order to build large systems consisting of thousands of processors, the broadcast and wired function facilities must be presumed.

If, as proposed above, a small number of signals are to connect every processor in parallel to the controlling task (or finite state machine), then all of the processors of the system are to be connected in a linear array. There must exist some other facility by which the objects communicate between processors. The simplest such connection is also a linear array or line such as an Ethernet [Metcalfe76]. Any other connection must be topologically more complex. Thus the connection of all the processor to a multi-conductor cable is on the same or on a simpler order than the network that must exist to connect the processors for normal communication.

The next difficulty with the global communication is that of fan-out and delay. Clearly, no logic technology provides the means to directly connect thousands of loads or sources to a single conductor as might be desired. However, due to the simple manner in which the processors and the controlling task communicate, a hierarchy of buffers can be constructed in a tree structure to limit the fan-out and fan-in of the components. If parts are used permitting a fan-out and fan-in of 16, then a system of 64K processors would have only four levels of buffers. If the buffers introduce a delay of 25

nanoseconds each and the propagation delay of the transmission lines between them is 900 nanoseconds (the delay seen across 600 feet of wire) then a very conservative estimate of the maximum delay between the controlling task and the processors is one microsecond. For the purposes of the garbage collection, this figure is so small it can be disregarded. The delay is proportional to the logarithm of the number of processors and will thus remain very low for even greater numbers of processors beyond the capability of current technology to package and power such systems.

The number of iterations of marking that must be executed by the processors before completion of the mark phase (denoted by the factor X above) must be investigated for large numbers of processors. To be a viable collection algorithm, the number of iterations must remain small as the number of processors is increased.

A number of processor configurations were simulated to test the effect of increasing the number of processors on the average time required to complete a garbage collection cycle. In each case the average number of objects per processor was maintained at approximately 25. All other factors in the model were held at the values described above.

Table 3-4

Simulation Results (with Message Polling)

Number of Processors	Average Cycle Time	Average Number of Repetitions per Cycle
1	0.52	2.0
2	0.78	2.4
4	1.19	3.1
8	1.62	3.3
16	1.91	3.2
32	2.33	3.3
64	2.72	3.5

Table 3-5

Simulation Results (with Message Interrupts)

Number of Processors	Average Cycle Time	Average Number of Repetitions per Cycle
1	.260	2.00
2	.290	2.85
4	.303	3.07
8	.309	3.27
16	.316	3.47
32	.333	3.96
64	.337	4.11

The average cycle time figures are computed in seconds. Because the timing in the simulator has no basis in the hardware of a real system, the absolute value of the numbers cannot hold much meaning. However, the relationship between these numbers is indicative of how the algorithm scales

as the number of processors in increased.

The simulator maintains no policy of locality to control the placement of objects among processors. Thus, in cases where a non-executing path of pointers and objects is strung across many processors and an object in the path is referenced by a non-garbage object, it may take several repetitions for the marking routines to follow the path through all the processors.

The average time required to complete a garbage collection cycle is observed to rise at a linear rate with the logarithm of the number of processors. The time to complete the cycle is a function of both the number of repetitions required in the cycle and the time required for all the processors to acknowledge messages, where polling is simulated. If processors are interrupted upon the receipt of a message from the controlling task, then there is essentially no skew between the processors and minimum delay in the acknowledgment of the message. The data tabulated for the simulation with interrupts are plotted in figures 1 and 2. The short vertical bars represent one standard deviation of variance about the average shown in the table.

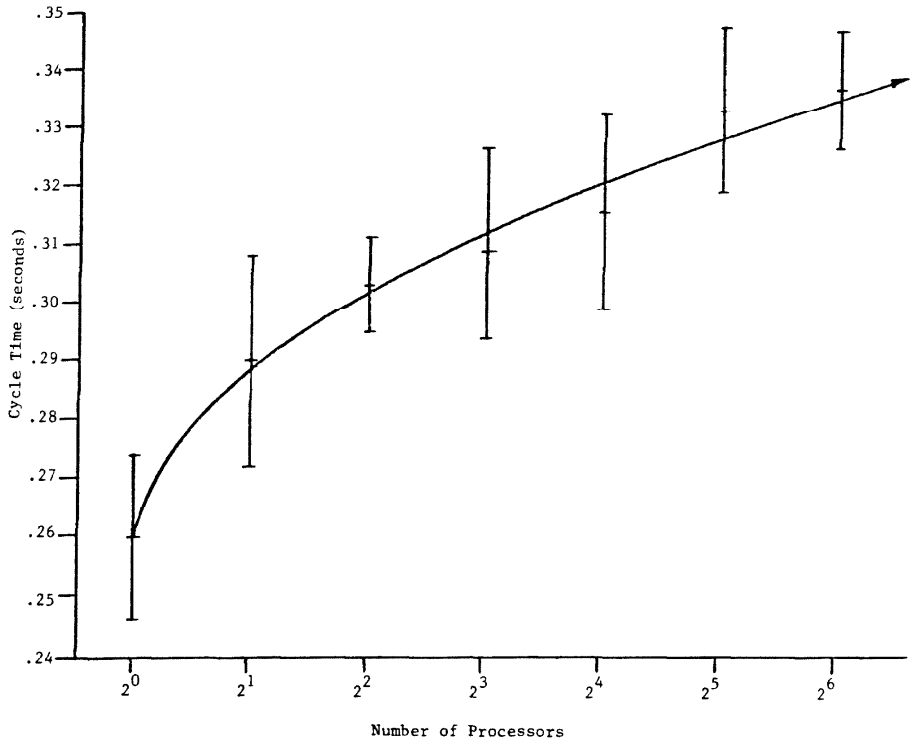


Figure 3-1

Cycle Time vs. LOG(Number of Processors)

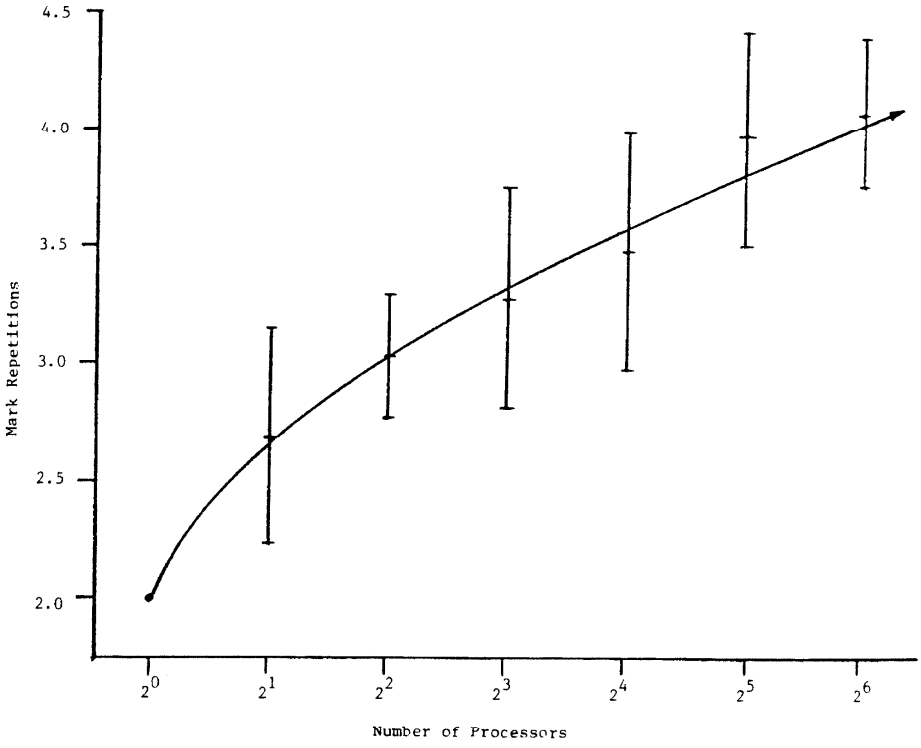


Figure 3-2

Mark Repetitions vs. LOG(Number of Processors)

The figures above show that in the worst case, the number of garbage collection cycles completed per unit time decreases linearly with the logarithm of the number of processors. The average number of repetitions required per cycle may grow linearly with the logarithm of the number of processors as well but there is some evidence that it may roll off to a value less than 5. In the best case, the average number of cycles per unit time also becomes a constant. Where the performance of a particular system falls between these two cases will be determined by the communication structure of the machine and the degree of locality present in the objects and processors. However, the worst case is seen to result in a performance of the garbage collector proportional to the inverse of the logarithm of the number of processors. If one merely extends the simulation figures to 64K processors, we see that the cycle time of the algorithm would be about 6.6, only 60% slower than 64 processors. The average number of objects per processor is presumed to remain constant, meaning that the 64K processor system contains 1,000 times the objects contained in the 64 processor system. If the relationship is as suggested by these numbers, the algorithm can be said to scale very well as the size of the system is increased.

3.9. Summary

A garbage collection algorithm has been presented which satisfies the needs of systems consisting of large numbers of processors. The algorithm has been demonstrated by simulating its operation. The unit of collection, the object, while a construct of programming languages, can be applied broadly to a wide range of systems, including conventional file systems and database systems.

The collection algorithm benefits from, but does not require, hardware communication facilities dedicated to the task of garbage collection. If these facilities are present, the speed of the garbage collection decreases in proportion to the logarithm of the number of processors, in the worst case. The introduction of techniques to improve the locality of reference among the objects in the processors will improve upon the already acceptable scaling characteristics of the algorithm.

The ability to collect garbage from data structures distributed among many processors in an efficient manner is a necessary ingredient to the use of very large distributed machines for general applications. Systems which provide for concurrency by connecting multiple processors to a single memory are necessarily limited in both size and performance. In the environment considered here, processors are the sole masters of objects resident in their private memories and communicate with other processors by passing messages through a communication network. The algorithm presented here will support the distribution of data and computation across a very large number of such processors without introducing more overhead computation than conventional collection algorithms require on existing systems.

Chapter 4

Interconnection Issues

4.1. Introduction

Considerable attention has been given to the analysis and development of interconnection schemes for distributed computer systems. Early work was directed at solving the problems of telephone systems [Benes65]. Given the computer architectures of the 1960's, much work has been and still is devoted to providing the means to connect multiple processors to multiple memory units [Lawrie73,Lang76]. The rapid evolution of integrated circuit technology has provoked interest in the interconnection of large numbers of micro-computers [Wittie76,81]. The existing work and analyses in this area are extensive. The work presented here is oriented toward a specific application not previously investigated. It is the purpose of this work to determine what the characteristics of several network topologies are, and how suitable they may be for the implementation of the object-oriented environment described in Chapter 2.

We consider here the interconnection facilities required to support a large number of physically small machines executing in an "object-oriented" environment. Objects and their messages are typically small but the rate of message production is usually high.

The processing nodes are substantial machines in their own right. To distinguish them from the type of processor found in systolic arrays

[Kung78] or a tree machine [Browning80], they are 1-2 MIPS (million instructions per second) and contain considerable private memory, perhaps 256K bytes to 1 Megabyte. Communication functions are handled by a specialized processor which has direct access to the processor's memory. Thus, the processor/memory node is not affected by messages passing through the node on their way to their destination. The instruction set and the internal architecture of the processor/memory node are of a high enough order to permit compilers to be written with some ease. In today's technology, such a machine can be implemented on a single printed circuit card. Over the next 10 years, advances in integrated circuit technologies might be expected to reduce it to a single chip or chip carrier.

Systems built around processing nodes, as described above, could contain thousands of nodes. Interconnecting large numbers of machines together such that they are able to adequately support the object-oriented environment places several requirements on the communication facilities.

(1) The communication facilities cannot be permitted to deadlock, despite cyclic topology or cyclic message flow. Deadlock is a condition where all or part of the network is unable to continue operation due to an unresolveable contention for occupied resources.

(2) As the system is expanded to include more machines, local communication must not be adversely affected. That is, increasing the size of the system must not slow down message traffic between neighboring nodes.

(3) The system must expand easily. The addition of processing nodes must not require the reconnection of existing nodes, nor can it require modifications to the nodes themselves.

(4) The system must be implementable. The cost of building the communications network must be commensurate with the number of processors in the system. For large numbers of processing nodes, the cost and difficulty of building the network must not become intractable.

(5) The average message delay exhibited by the system should be of the same order as the delay involved in a procedure call. Since we cannot expect all programs to be highly concurrent, the delay in sending messages must be kept on a par with a procedure call in a conventional computer. Thus, highly sequential programs will still run with acceptable performance. It is desirable that communication delays be balanced with computational delays in the processors to avoid bottlenecks for concurrent programs.

(6) The routing of messages must not require the presence of global topology information in each machine. The communication processors must not require a map to route messages. Such information would grow with number of nodes in the system and become too large to contain in each processing node.

Given the above restrictions, certain interconnection schemes can be eliminated immediately as candidates. Complete interconnection has obvious desirable communication properties but is unimplementable except for very small numbers of nodes as it requires $O(N^2)$ communication links. Likewise, a crossbar switch arrangement [Pippenger75] is undesirable, since it too becomes impossible to implement for large N, requiring N^2 switching elements. A star configuration, while it appears to require only one link per node with a special processor at the center of the star, actually requires a crossbar switch at the center of the star if it is to function at an acceptable

performance level.

At the other end of the spectrum, there are schemes such as Ethernet [Metcalfe76]. While such a network is quite simple to implement, the message delay seen between any pair of nodes will clearly increase as more nodes are added, due to the increased contention in the system. This characteristic of contention busses does not rule out their use in a hierarchy or some other organization of multiple busses. The hypercube described in [Wittie76] is an interesting case.

Another interesting topology is the cube connected cycle [Wittie81]. This scheme employs rings of nodes at the vertices of a Boolean N-cube where there are $\log_2 N$ nodes in each ring (N is the number of vertices in the Boolean N-cube). Each node in the ring connects to its two neighbors in the ring and also makes one connection to another vertex. The nodes would thus have a fixed set of three connections, however, expansion of the system requires inserting nodes into all of the existing rings, as well as adding vertices. This massive rewiring of the system makes the cube connected cycle undesirable at the outset.

Several interconnection topologies are investigated here as representative of some class of structures. Specifically, the chordal ring, the tree, the toroidal mesh and the Boolean N-cube have been studied. Each have been extensively simulated with variations in parameters, such as, differing link data rates, queue lengths, number of processing nodes, etc. The simulated message traffic was also varied. To permit the various topologies to be compared on an equal basis, a model of message locality is presented which is independent of the dimensionality of the interconnection. Use of this model permits the application of message traffic of equal locality

to systems in which the measures of distance are widely different.

The simulation models and their results are discussed and conclusions are drawn about the appropriateness of these interconnection structures for use in an object-oriented machine.

4.2. Interconnection Topologies and Queuing Models

Four topologies have been simulated using a packet switching model of message handling and using queues at various points to smooth out short term congestion. To compare these various topologies on an equal basis all factors other than the interconnection strategy are treated identically in the following analyses.

Messages of arbitrary length are broken into fix sized packets for transmission through the network. A fixed size packet permits hardware queues and buffers to be of limited size.

Communication links are all assumed to be bit serial and full duplex. Unless otherwise stated, it is assumed that the bit serial links have a bandwidth of 20 Megabits per second. The systems under investigation here are tightly coupled and physically compact to improve wireability and communication speed. The maximum length of the links in such systems should be well below 50 feet. Data rates as high as 50 Megabits per second are achievable over such distances with conventional interconnection media. Full duplex operation would require that each link consist of at least two conductors, one for communication in each direction. Communication links must be limited to one or two conductors, otherwise implementation of large systems is made proportionally more difficult.

At each processing node, there exists the aforementioned processor and its memory as well as a communication processor with a high bandwidth, parallel interface to the memory. The serial communication links are part of the communication processor as are any associated packet queues. Figure 4-1 is a block diagram of the model used to represent a processing node.

There exists a queue (FIFO) memory for each output communications link. The queue is assumed to have a zero fall-through time and is of a fixed size. Unless otherwise stated, the queue size of all queues is four packets. There is also a queue between the object processor and the communication processor for outgoing packets. Incoming packets are assumed to always have a place in the memory.

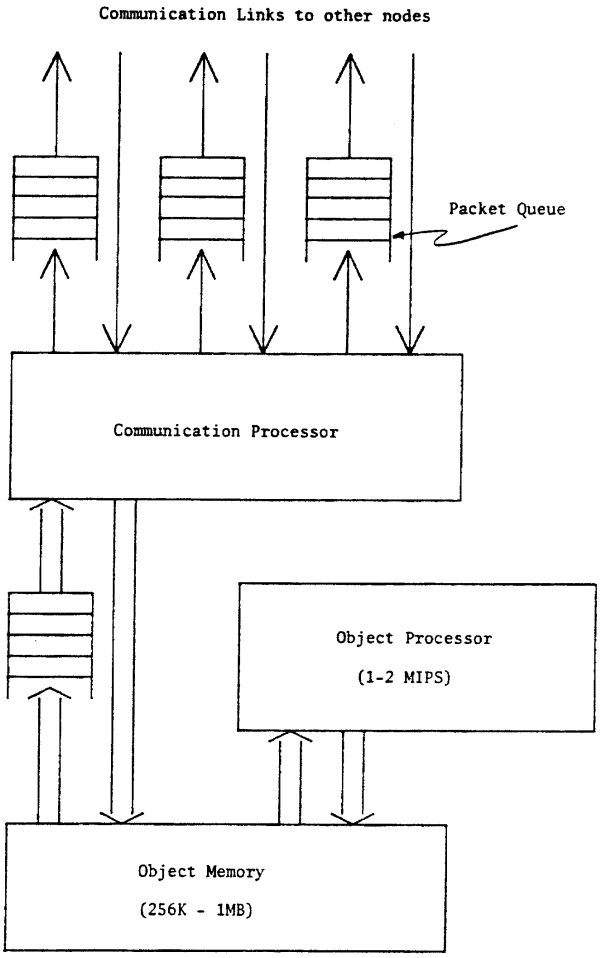


Figure 4-1

Processing Node Block Diagram

The packet size for these networks was chosen to be 256 bits of data. Addresses, error codes, routing information, etc. are part of the packet but are not included in the packet size. This size was chosen to avoid excessive fragmentation and message assembly overhead and to give the links an acceptable duty cycle. Message passing languages are usually characterized by a large volume of small messages. The parameter space over which these networks have been simulated reflect these characteristics.

Message traffic is described by several parameters. The average message length and a standard deviation, the average amount of computational time spent between the generation of messages and its standard deviation and the locality with which the destinations of the messages are chosen. The parameterization of locality is discussed in a succeeding section. Based on experience with Simula, an average message length of 768 bits with a standard deviation of 256 bits was chosen. Thus the average message is three packets long, with a standard deviation of one packet.

The rate at which messages are generated by each processor was chosen to be high for two reasons. First, message passing languages tend to send messages with a frequency approaching the rate at which other languages execute procedure calls. The simulation of this type of operation requires a correspondingly high rate of message production. Also, to obtain a good comparison of the the topologies and to find their limitations, it is necessary to saturate them. The rate of message production used in simulation here is one message every 30 microseconds with a standard deviation of 15 microseconds. For a processor such as the Motorola 68000, this rate corresponds to one message per 80 or 90 instructions.

4.2.1. Tree Connection

The use of tree structures to connect machines is very common [Horowitz81]. It has several virtues that make it attractive. It has a planar topology, thus guaranteeing that it can be implemented without unusual effort. The routing of data from one node to another is accomplished with simple algorithms local to the nodes. It is also deadlock free because it contains no closed loops, preventing the possibility of cyclic dependencies that would constitute deadlock.

The tree structure used here puts all the processing nodes at the leaves of the tree. The remainder of the tree serves as a facility for communication only. Nodes in the tree connect with their parents by mean of a communication link as previously described. The links have a queue of packets associated with messages traveling in each direction. Figure 4-2 illustrates the tree structure.

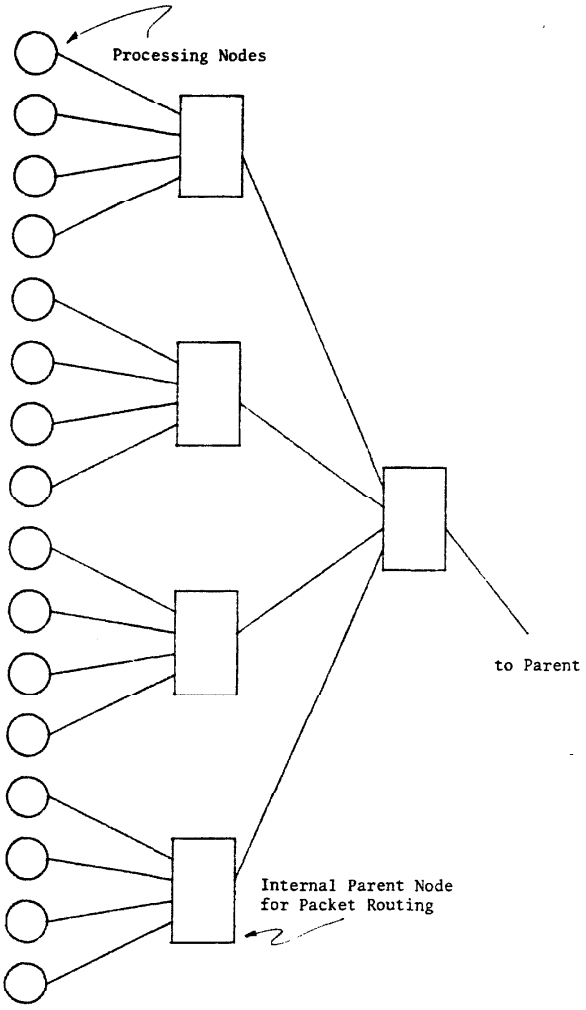


Figure 4-2

Tree Connection

A message or packet that is to be passed from one leaf node to another must be forwarded to the closest common parent and then back towards the leaves to the destination. There is one and only one such path between any pair of processing nodes. Other tree-like structures have been proposed, usually containing communication paths between nodes at a given level. The X-Tree [Despain78] and [Harris77] both propose "horizontal" connections between branches of the tree. The introduction of these paths produces cycles in the interconnection graph and may thus introduce the possibility of deadlock. This issue will be discussed in a succeeding section.

The branching ratio of the tree has a large effect on the communication characteristics of the tree. A small branching ratio, such as 2, maximizes the number of links a message must travel to reach its destination. A large branching ratio reduces the height of the tree but as the number of links at each parent node grows, so must the complexity and congestion of the node grow. The extreme case, where the branching ratio equals the number of processing nodes, produces the star structure with its complex central switch.

For the purposes of this analysis, a branching ratio of 4 was chosen. This results in communication nodes with 5 connections making them fairly easy to build. The height of a tree containing 64K processing nodes at the leaves and having a branching ratio of 4 would be 8. The longest communication path would then be 16 links. This distance is the same as the longest path found in a Boolean N-cube of the same size. In general, the maximum communication distance found in the tree is given by the expression $2\log_b N$ where b is the branching ratio and N is the number of processing nodes.

Clearly, a branching ratio of 4 encourages local communication among groups of 4 or fewer processing nodes. Whatever the branching ratio, the tree structure imposes an increasingly severe penalty in situations where the locality of communication involves more processing nodes than the branching ratio.

Packets are routed easily in the tree. If a parent receives a packet with a destination corresponding to one of its descendents, it sends (or queues) the packet using the link which is a branch to that descendent. If the packet is not for one of its descendents, the node sends the packet to its parent.

Congestion can be expected in the parent nodes if the production of nonlocal messages by all the parent's descendents exceeds the bandwidth of the link to the parent's parent. The packet queues between the parent and its descendents will fill up with packets until the processing nodes are made to wait before sending another message. The effective rate of message production will be reduced to match the rate of message consumption at the bottleneck by reducing the utilization of the processing nodes. This tradeoff between processor utilization and communication bandwidth may occur in all communication networks and is not limited to the tree structure.

4.2.2. Chordal Ring Connection

The Chordal Ring connection is one in which N processors are connected in a single closed loop with the addition of connections from each processor to another processor a fixed chord length along the circle. Figure 4-3 shows a chordal ring connection of 16 processors. An analysis of chordal rings can be found in [Arden81], where it is shown that the maximum length of an optimum path between two nodes in a properly constructed chordal ring is $O(\sqrt{N})$.

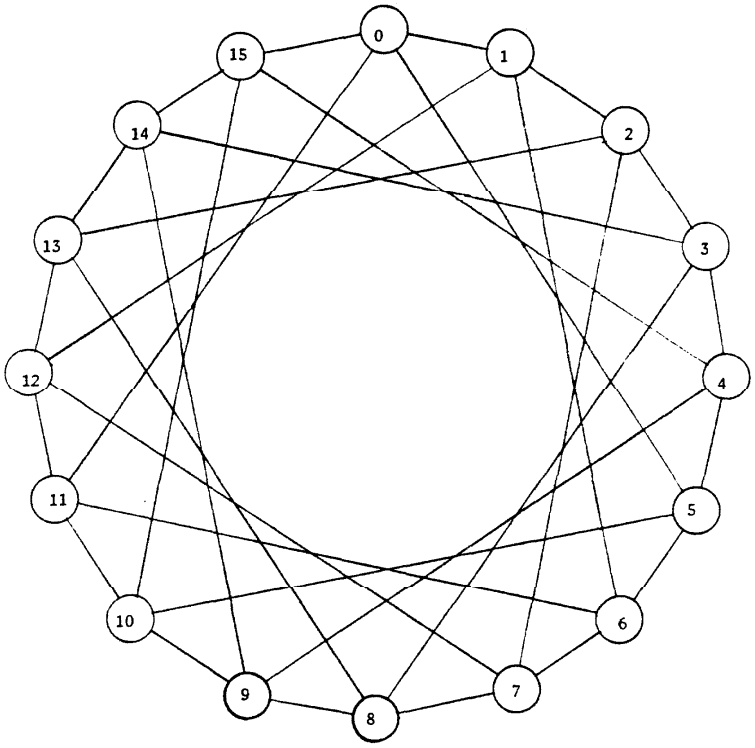


Figure 4-3

Chordal Ring Connection

The chordal ring, as used here, is made of processing nodes connected by queued links as previously described. The chord length is chosen to be $\sqrt{N}$ to approximate the conditions of [Arden81]. In addition, as can be seen in Figure 4-3, each node has 4 connections, two are part of the ring and 2 are chord connections. This addition gives the chordal ring a lower effective diameter and gives it the same degree connection as the mesh connection described in the next section. The degree of connection (4) is close to that of the tree connection (5) as well, making the comparison of the chordal ring with the other connection strategies more equitable.

The links in the chordal ring are bi-directional permitting messages to move in either direction around the links and chords. The routing algorithm attempts to send packets to their destination by the shortest path. When a packet is received at a node which is not its destination, the node decides which of the other three connections to the node would send the packet closest to its destination. The message is then queued for the selected link. If the selected link queue is full the second choice is used, providing it is not full.

The chordal ring is a simple connection strategy of low degree. Though it is not planar, it would not be difficult to implement for even very large numbers of processing nodes. Since it does have a cyclic graph, there is a probability that it could deadlock unless other measures are taken. The simulation results show how suitable this topology is for the needs of an object-oriented machine.

4.2.3. Toroidal Mesh Connection

Two dimensional mesh connections are constructed by arranging processing nodes at the vertices of a grid in a plane and connecting each

node to its four nearest neighbors. Hexagonal arrays are constructed in a similar manner. These types of connections have the advantage of being inherently planar and of fixed degree. They are proposed for use in various applications where the topology of communication can be made to match the topology of the interconnection such as [Martin81] and [Kung78]. Meshes of higher dimensions can be constructed as well [Wittie81] with increased communication capabilities but incurring increased implementation costs.

Here we consider a two dimensional mesh with the edges of the mesh wrapped around to form a toroid. This avoids difficulties with the boundary conditions at the edge of the mesh and maintains a fixed degree of 4. The edges are wrapped around without the twist introduced by [Martin81]. The toroidal nature of this connection is non-planar but can be easily provided for in an implementation. One additional channel in each dimension and between each row and column of nodes must be provided to accommodate the wrap around connections. An illustration of the toroidal mesh is shown in Figure 4-4.

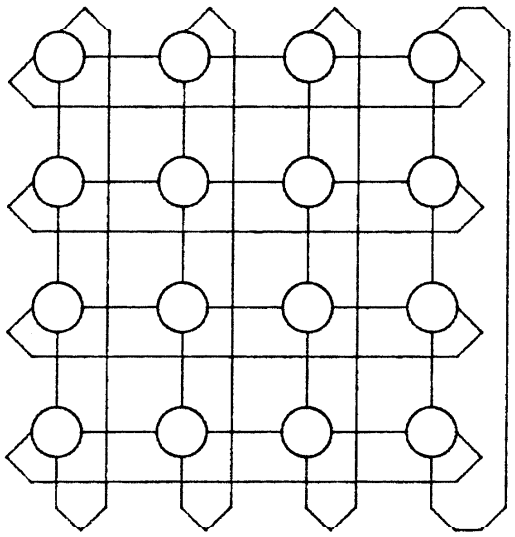


Figure 4-4

Toroidal Mesh Connection

The maximum distances that must be traveled by packets in the mesh are $O(\sqrt{N})$ since they must travel a Manhattan path to their destination. This is the same result as found by [Arden81] for the chordal ring. In the configurations to be simulated, the mesh is always square and each node has 4 queued links as described for the other topologies. The routing of packets in the mesh is determined as follows:

(1) The node addresses are viewed as coordinates and the distances to the destination in directions north, south, east and west are determined.

(2) In the order of smallest distance, each queue in the associated direction is polled and if space is available, the packet is queued there. If no space is found in the queue, the direction with the next larger distance is polled.

(3) When all four queues have been polled and no space has been found, the process repeats until the packet is queued.

This procedure guarantees a minimum length path is used if there is no traffic congestion. Where congestion occurs, an attempt is made to route the message around the congested area. It is even possible for a packet to be sent back along the same link it was sent if congestion is severe. If this happens, a less congested path may be found for the packet.

Like the chordal ring connection, the mesh connection graph has cycles giving it the potential to deadlock. Like the tree, the mesh connection clearly favors local communication traffic with groups of 4 or 5 processing nodes. As traffic becomes less localized, more congestion will occur in the queues and the probability for deadlock will increase.

4.2.4. Boolean N-cube Connection

The Boolean N-cube interconnection is a multi-dimensional, variable degree strategy where nodes are connected to their neighbors in N-space. The simplest definition of this interconnection is to first number all of the nodes sequentially starting at 0, then connect all pairs of nodes whose numbers have a Hamming distance of 1 (those whose numbers differ by only one bit). Figure 4-5 shows a Boolean N-cube of 16 processors. The definition requires that each processor have $\log_2 N$ connections, where N is the number of processing nodes in the network. The Boolean N-cube can be considered an extreme case of the toroidal mesh, where only two nodes are permitted in each dimension and the number of dimensions is increased to accommodate additional nodes.

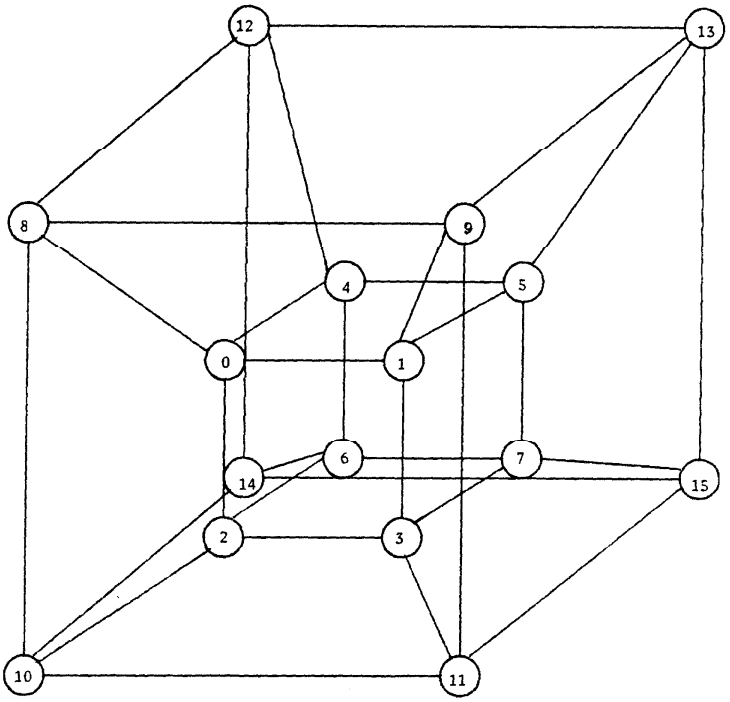


Figure 4-5

Boolean N-cube Connection

The Boolean N-cube has long been an interesting interconnection scheme for various dedicated computations [Pease75]. It has been shown to be functionally equivalent to several other schemes, such as, the perfect shuffle [Stone71,Lang76], the Omega network [Lawrie73] and Benes' re-arrangeable network [Benes65] by [Parker80]. Sullivan [Sullivan77] first advocate its use to interconnect autonomous processing nodes. Because the Boolean N-cube is ubiquitous, simple and exhibits some obviously desirable properties, it is investigated here. It is also clear that for large numbers of numbers of nodes, the Boolean N-cube becomes increasing difficult to implement. This issue will be taken up in a succeeding section.

The Boolean N-cube clearly contains cycles in its graph, making it subject to deadlocking. The next section will shown how packets can be routed to avoid deadlock with some loss of generality and bandwidth.

Given the same queued links used in the preceding interconnect strategies, each node in the Boolean N-cube connects to one neighboring node in each of the other dimensions. In the general case, routing the packets to their destinations by the shortest path is quite simple. Owing to the original definition of the structure, each of the links to a given node connect to all the other nodes in the network whose addresses differ from its own address by exactly one bit. Each link can then be associated with that bit of the address which differs in the nodes it connects. When a node receives a packet it performs an exclusive-OR operation between its address and the address of the packet's destination. If the result is zero, then the packet is at its destination. If not, the packet may be sent across any link corresponding to a "one" bit in the result of the exclusive-OR. In this way, successive nodes route the packet, changing one of the differing bits until

the destination is reached. Where there is a choice of more than one link on which to send the packet, the link whose queue is least full is selected to avoid congestion and balance the traffic load.

The longest path between any two nodes in the network is $O(\log_2 N)$. This is a better characteristic than is found for the toroidal mesh or the chordal ring whose maximum distances are $O(\sqrt{N})$. It is comparable to the tree in this respect but it should be noted that in the tree there is only one path between any pair of nodes, and parts of that path are heavily shared by other paths. In the Boolean N-cube, for a pair of nodes with Hamming distance m, there are m! paths of length m between them [Sullivan77]. There are longer paths as well but these are not considered here. It is important to note that, in the Boolean N-cube, the farther a packet must travel, the more paths there are for it to take.

4.3. Deadlock

Deadlock cannot be tolerated in any system unless its probability of occurring is less than the probability that the system may experience a hard fault that would render it unavailable. With the exception of the tree connection, all the interconnection systems investigated show some danger of deadlock. The tree connection is deadlock free because its graph contains no cycles. The probability of deadlock in other networks is a function of the size, queuing, topology and message traffic in the system. Without *a priori* knowledge of the message traffic, the probability with which deadlock will occur can not be determined.

The mechanism of deadlock requires several conditions. It requires at least one cycle in the graph of interconnected queues. Also, it presumes that a packet with any destination may appear in any queue. It is this last

characteristic that the system designer has control of, though, in some cases, exercising this control can be otherwise undesirable. The following is a proof of the existence of deadlock given the above assumptions.

(1) Extract any loop of interconnected queues from the system as shown in Figure 4-6.

(2) Assume that all the queues in the system are filled with packets. Assume further that the queues are of a fixed and finite size (as they must be in a real system).

(3) Assume that all the packets in the queues have a destination node within the loop but one other than the node they next encounter (the node the queues point at). If packets cannot change order in the queue it is only necessary to assume that the first packet in the queue (the one that is to be removed next) has a destination beyond the next node.

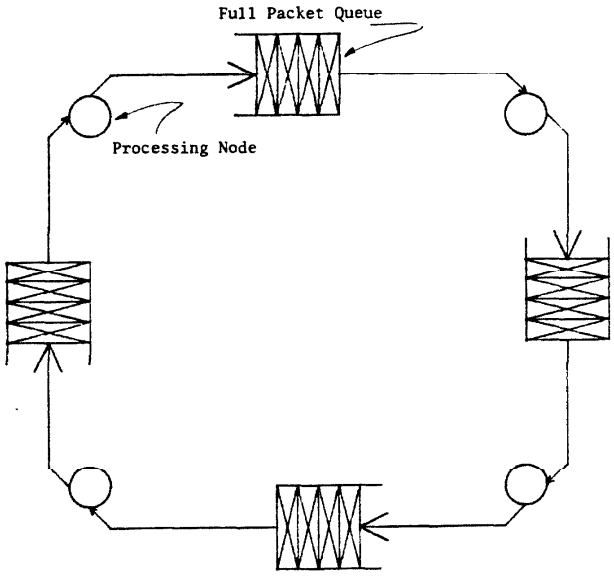


Figure 4-6

A Deadlocked Loop of Communication Links

The above situation is that of deadlock. The following arguments prove that the nodes and queues in the loop will never be able to change their state and this state thus constitutes deadlock.

(1) To remove a packet from a queue, a node must either consume the packet or place it in another queue. To consume the packet it must be destined for that node. The situation is defined such that packets in the queues are never destined for the next node and thus cannot be consumed. The packet cannot be moved to any other queue since they are all filled.

(2) Thus, if no packets can be consumed, and no packets can be moved, then no new space will ever become available in a queue.

(3) If no space ever becomes available in a queue, no node will ever be able to move a packet, etc.

If the above state of the system is then deadlock, we must only show that it is a reachable state to prove that there is a probability of deadlock. It should be noted, that the tree can never have the above state, since by its definition, the queues pointing at the leaf nodes contain only packets destined for the leaf nodes. Thus, these packets are always consumable and the queues pointing toward the leaves are always emptying.

To reach the described state, the following could happen:

(1) From a condition in which all the queues are empty, meaning there is no traffic in the system during some interval in time, all the nodes begin producing messages.

(2) The rate of message production exceeds the data rate of the links for a long enough time such that messages fill the queues.

(3) If all the messages are destined for nodes more than 1 link away, then the queues are filled with messages of the type described above and the system has reached the deadlocked state.

While the foregoing may seem somewhat contrived, it is only one state and procedure constituting deadlock. If any portion of a system deadlocks, it will increase the probability that other parts will do so by causing queues to remain filled and immobile that would otherwise be available to the system. In any case, if any part of the system deadlocks, even two nodes, the failure is unacceptable.

Deadlock is not easily detected in the system and if detected it may not be possible to unravel the system without some loss of state. It is necessary that if deadlock occurs that it occur so infrequently as to be unnoticed when compared to other failures of the system.

4.3.1. Deadlock Avoidance in the Boolean N-cube

Given that the tree may not be the most desirable topology for interconnecting nodes, it is possible to control message traffic in the Boolean N-cube to prevent any possibility of deadlock. Equivalent controls may exist for the chordal ring and the toroidal mesh. In general, if message traffic can be controlled or restricted such that packets can always be consumed, deadlock will not occur. Also, if congestion can be detected, processing nodes can be prevented from generating more messages until the congestion is relieved again preventing deadlock. Both the toroidal mesh and the chordal ring can be modified from the general form shown here to a specific system with provisions for avoiding deadlock. The Torus machine [Martin81] uses a toroidal mesh connection and avoids deadlock by restricting communication between processing nodes to a fixed pattern.

Sullivan [Sullivan77] presents a message routing procedure for nodes in a Boolean N-cube which avoids deadlock, though he does not recognize this characteristic in this reference. The procedure is less general than that already presented for the Boolean N-cube. It permits messages to be routed between any pair of nodes but restricts the flow to one particular path. The procedure is restated here.

(1) Exclusive-OR the node address with the destination address of the packet. If there is no difference, consume the packet.

(2) If there is a difference, send the packet to the link corresponding to the leftmost differing bit of the two addresses.

The following is a proof that this routing procedure renders the Boolean N-cube deadlock free.

(1) Packets in the queue of any link corresponding to the rightmost address bit must always be destined for the next node and can therefore, always be consumed by that node.

(2) Packets in the queue of any link corresponding to the second rightmost bit must either be destined for the next node and can be consumed or must go into the queue that is to the right of the link's position. Item 2 of the routing procedure above insures that any differing bits between the current node address and the destination address must be to the right of the bit corresponding to the link the packet was received from. The rightmost queue is the only queue that is logically to the right of the second rightmost queue. If it is emptying, as in Item 1 above, then there will be a place to put packets from this queue causing it to empty as well.

(3) Apply the above argument to each successive bit of the address until the leftmost bit is reached, proving that the queues corresponding to the leftmost bit will always empty.

(4) If all the queues will eventually empty, then deadlock cannot occur.

Therefore, by using Sullivan's routing procedure, a Boolean N-cube connection can be using without fear of deadlock. The single paths used by the packets will be evenly distributed over the system. However, where local congestion occurs, particular packets will be unable to circumvent the congested queues. This loss of generality can be expected to reduce the performance of the Boolean N-cube from that achieved by original routing algorithm presented here. If deadlock can be avoided in the mesh or ring connections by restricting the routing of messages, their performance will be reduced as well, because the number of choices available in routing messages will be reduced.

The more restrictive routing procedure avoids the problem shown in Figure 4-6 by controlling the destinations of packets that may be found in the various queues. Using this procedure, no loop of queues can exist where none of the packets can be consumed by the attached node.

4.3.2. A Non-Queued, Deadlock-Free Interconnection Structure

Deadlock arises in queued systems where conflict occurs over the allocation of resources, in the form of queue slots. Once a queue slot has been taken by a packet it cannot be released to another packet until the first packet has been properly disposed of. In the previous section, deadlock was avoided by insuring that queues always emptied, providing a constant and guaranteed turnover of resources. Another method of avoiding deadlock is to not permit resources to be allocated until enough have been reserved

for the entire operation. In a communication network, this means reserving all the communication links required to send a message, send the message and then release the links. The reservation of resources must be done in such a way as to prevent deadlock as well.

In this section, we present a scheme of reserving and releasing links in a Boolean N-cube connection that is deadlock free. This system contains no concept of packets. Once an entire path, consisting of one or more links, is reserved to the message's destination, the entire message is sent *en masse*. The nodes pass the serialized data through from one link to the next with little or no buffering permitting the message to be transmitted at the full bandwidth of the link and with a very small propagation delay.

The hardware at each node to connect one link to another, or to the processing node, consists of a small crossbar switch. The size of the crossbar switch is $1+\log_2 N$ by $1+\log_2 N$ where N is the number of processing nodes in the system. The extra connection is for communication to and from the processing node itself. The data paths in the crossbar have a width of 2 for full duplex communication. For large N, such as 64K, the crossbar switch in each node is well within an acceptable size. For 64K processing nodes, the crossbar switches must have on the order of 272 switching elements. Associated with each link, there must exist control logic and sufficient state to implement the reservation protocol and make connections in the crossbar switch.

There are several definitions and rules that must be obeyed by the links as they are manipulated. A link must be in one of the following states:

(1) UNRESERVED In this state, any path may reserve the link causing it to become RESERVED.

(2) RESERVED In this state, the link is reserved to a given path. Another path may take the link and reserve it for itself if the priority of the new path is greater than that of the path to which the link is currently reserved.

(3) COMPLETED In this state, the link is assigned to a path and cannot be reserved by any other path. Only the path to which the link is assigned may change its state to UNRESERVED.

To establish a complete path to a destination node, the originating node begins by attempting to reserve a link corresponding to a bit which differs with the destination address. In reserving a link, the node may be successful and the same process is repeated at the next node until the destination is reached. If the link is currently reserved, the node will wait or choose another link for reservation, if there is a choice.

Once a link is reserved, it may yet be taken from the path by a path of greater priority. The priority of paths is first determined by the number of links they contain. The longer a path gets, the higher the priority of all its reserved links. If a longer path attempts to reserve a link reserved to a shorter path, the link will be taken from the shorter path and reserved by the longer. The shorter, now broken, path is made to dissolve and reattempt a new connection. If two paths attempting to reserve the same link have the same length, the link will be reserved by the path originating at the highest numbered node. In this way, no two paths have the same priority. Unique priorities prevent the paths from deadlocking over contention for the links.

When the last link in a path is reserved, all the links in the path become COMPLETED. The path is then established and used for the duration of the message transmission. After the entire message has been sent, all the links

in the path are released by sending them to state UNRESERVED.

This procedure is used by all the nodes in sending messages. Since none of the the data in the message is stored in the network, the breaking of paths involves no loss of data. As an added advantage, the message is sent in order, eliminating the need for the reassembly of packets as required in queued systems. The system does not deadlock, because one path will always win any contention. Congestion of the network will result in many paths being broken before they are completed, but since they are able to retry until they do achieve a complete path, the system exhibits liveness.

This type of Boolean N-cube was also simulated to compare it with the queued networks. In these simulations, each nodes has a queue of 4 messages produced by its processor, but there are no other queues in the system. To change the state of a link, the simulations require enough time to transfer 64 bits to the link, such that link reservations occur in a finite time. The link data rates and message traffic used are the same as those used in the queued networks.

4.4. A Distance-Independent Measure of Locality

To make meaningful comparisons of performance among networks of differing topology, the message traffic used to test them must have no topology dependent notion of distance. In each of the topologies discussed here, the distances seen from one processing node to another depend on the particular network. A model of message traffic is required which is independent of the network in question but can be applied to any network in the same way that a program or set of objects can be executed on widely different machine structures.

To start, we introduce the concept of a "neighborhood". From the point of view of a single processing node in a network, its neighborhood is a set of other processing nodes with which it will communicate with a probability greater than some threshold T. The size of the neighborhood is a measure of message locality. If the neighborhood size was the same as the number of nodes in the system, then traffic in the system could be said to be uniform. Every node would have an equal probability of communicating with any other node. For small neighborhood sizes, traffic can be said to be very localized where the probability that a node communicates with one of its neighborhood set is much higher than for other nodes. The size of the neighborhood is denoted by α.

For given message traffic, the probability that a given node communicates with any selected node can be determined. The nodes can then be arranged by decreasing probability. The neighborhood is ideally the first α nodes in the ordered list, where α is to be a measure of locality. The probability that each of the first α nodes will communicate with the given node will be greater than or equal to the threshold T. The list can be approximated by a geometric distribution where:

$$f(x) = \frac{1}{\alpha} e^{\frac{-x}{\alpha}}$$

The dependent variable x is the position in the list and $f(x)$ is the probability of communication with node at that position. The constant α can be thought of as the approximate size of the neighborhood. The geometric distribution has the desirable programming property of having a simple, closed inverse. Also, its mean and variance are conveniently given as α and α^2, respectively.

In the simulations, the message locality is a parameter. The parameter a is the desired size of the neighborhood and is used, as follows, to generate appropriate message traffic. A number X is picked at random in the interval between 0 and $\frac{1}{a}$. The inverse of the geometric distribution is used to find what position p in the list of nodes is represented by this probability:

$$p = -a\,Log_e(aX)$$

At this point, p represents a randomly chosen number with the distribution of the desired message traffic. It remains to select a corresponding processing node in the network being simulated. This process converts the number p to a corresponding distance in the network being simulated. That is, a distance d is chosen which is the number of communication links that must be traveled in the network to reach any one of p processing nodes.

This transformation is different for each network topology. For each of the topologies in question, the follow relationships hold, describing how many nodes R can be accessed by traveling *exactly* l communication links, where N is the total number of nodes in the network:

(1) Boolean N-cube ($0<l\leq \log N$)

$$R = \frac{(\log N)!}{l!(\log N - l)!}$$

(2) N-ary Tree (where b = branching ratio and $2\leq l \leq 2\log_b N$)

$$R = (b-1)\,b^{\frac{l}{2}-1}$$

(3) Two Dimensional Array (assuming large N and no boundaries)

$$R = 4l$$

(4) Chordal Ring (assuming large N)

$$R = 4l$$

The above functions are integrated with respect to l (the number of links) to produce a table. The table can be indexed by p to find the corresponding entry which is the distance d, in links, that must be traveled to access p nodes.

Of the specific nodes that are found to be exactly distance d from the node that is to send a message, one member of the set is chosen at random. The chosen node is then sent a message in the simulated system. As all the nodes in the network exhibit the same behavior and select the destinations of their messages by the same method, the overall message traffic has a locality determined by the original parameter α.

In the simulation results, references to "Traffic" indicate the level of message locality as set by α. A small α, in the range of 3 to 5, is highly local traffic. An α of 12 or more may be regarded has substantially non-local traffic for systems of less than 100 processing nodes. The parameter α has no effect on message length or frequency, it affects only the destinations of messages.

The intent of this model of traffic locality is that it be used to represent communication requirements of an object-oriented program. Since it is possible for the same program to run on machines of different topology, the parameter α has been made independent of distance. The simulation results are normalized by the use of α. The use of the parameter α in the simulated traffic of each system gives an indication of how each will react to the same class of application programs. The group size, as represented by α, describes a homogeneous program execution. Real programs may exhibit nonhomogeneous communication by having objects with widely different characteristics. The placement of objects in processors can cause

nonhomogeneous communication among the processors. Such programs might be better characterized as a composite set of several group sizes rather than one group size. The following simulation results are based on homogeneous programs whose locality of communication is characterized by a single parameter α.

4.5. Simulation Results

For each of the interconnection strategies described, a simulation program was written using the Demos simulation package [Birtwhistle79]. The size of the network, the message traffic, the queue sizes and the data rate of the links were all parameters to the programs. In all, six network types were simulated as listed in Table 4-1.

Table 4-1

Names of Networks Simulated

Name on Graphs	Network Type
ARRAY	Square Toroidal Mesh
TREE	Tree with Branching Ratio 4
RING	Chordal Ring
NCUBE	Unrestricted Boolean N-cube
ECUBE	Restricted Boolean N-cube
SCUBE	Non-queued Boolean N-cube

The default parameters used in the simulations have been previously described. Unless otherwise stated, the following parameters were used:

(1) link data rate: 20 Megabits per second

(2) packet size: 256 bits

(3) maximum number of packets in each queue: 4

(4) distribution of message lengths: Normal distribution, mean 768 bits and standard deviation of 256 bits

(5) distribution of message intervals: Normal distribution, mean 30 microseconds and standard deviation of 15 microseconds

The simulated systems were "run" for a simulated period of 2 milliseconds to cause them to reach an equilibrium. At this point, the data collection facilities of the simulators were reset and the system was simulated for 8 milliseconds. For most systems, several thousand messages would be produced by each processing node and transmitted within this time period. During this phase of execution, various statistical measures were taken of the network performance. A characteristic listing of the results of a simulation is found in Appendix B. The following is a list of those measures that were selected as important to this application. These quantities are displayed graphically in Appendix A.

(1) *Average message delay* The mean time between the production of a message by a processor and its completed receipt at its destination. This is an overall measure of how good the network is at getting messages to their destination with some given traffic density.

(2) *Average message delay of messages traveling distance 1* This is the mean delay seen by messages traveling through exactly one communication link. This delay is intended to measure how good the network is at moving very local messages to their destinations.

(3) *Average packet delay* The mean time required for a packet to reach its destination. This number is clearly related to the average message

delay but removes the influence of the message length and the situation where one or more packets delay the entire message. In graphs showing this function, traces marked as SCUBE are plots of the time required to complete an unbroken path to the destination. The SCUBE has no packet concept.

(4) *Processor Utilization* This quantity is a measure of what percentage of time processors were not idle, waiting to place their next message in a queue. When the queue between the processor and the communication node is full the processor is made to wait until space for a packet is available. For some level of message traffic, this percentage is a measure of the bandwidth of the network as a whole, that is, its ability to keep up with processor message production.

(5) *Port Utilization* This quantity is the percentage of time the communication links are actually being used to transmit data. This number is their overall duty cycle.

The parameter space over which the networks were simulated was narrowed to a specific area. Computing costs and address space limitations prevented more than 96 processing nodes from being simulated in all but the SCUBE configuration. To investigate the effects of scaling, the size of the networks the range of 8 to 96 processors was heavily simulated. Also, the effects of varying the link rate from 1 to 50 megabits per second were simulated, as well as, average message lengths from 3 to 8 packets. Message locality, as measured by the α parameter, was varied from 3 to 24 to determine how well the various networks responded to traffic of varying locality.

To measure the worst case traffic, the special case of uniform traffic was simulated. Uniform traffic is defined to be the case where any message has an equal probability of being sent to any node in the network. This can be considered the worst case, because if objects were scattered at random over the nodes, this type of message traffic would be the result. Surely, any algorithm used to distribute the objects in the system would do no worse than this.

Appendix A contains the bulk of the simulation data in graph form. Here, the parameter space of 8 to 96 processors and α=3 to α=12 and uniform traffic is explored. Interspersed within this chapter are some of the more significant results. Appendix B contains the complete set of data output for one simulation. The network simulated was that of the NCUBE with 64 processing nodes. The statistical measures and histograms are typical of the data produced by the simulation programs.

In Figures 4-7 and 4-8 the effect of message locality on the message delay and processor utilization can be seen. It is quite clear that the RING and TREE connections suffer severely as the locality parameter α is increased. It is notable that the NCUBE connection actually improves or is constant with decreased locality. This is explained by the fact that the farther a message travels in the NCUBE, the more paths it has to choose from. As the parameter α is increased, a larger percentage of the message traffic is able to benefit from the increased number of paths. The ECUBE, which restricts each message to one particular path, is seen to lose performance slowly as messages are unable to avoid congestion. The SCUBE suffers more severely since, less locality causes greater numbers of links to be tied up in each message transfer. The best performers here, the NCUBE

and ECUBE provide delays of less than 100 microseconds and are able to support more than 75% of the processor message production under the conditions stated.

The effect of varying link data rates on performance is seen in Figures 4-9 and 4-10. The average message delay does not fall below 100 microseconds in the RING, TREE and SCUBE until the data rate of the links exceed 35 Megabits per second. As serial data rates above this figure are difficult to implement, this limits the choices of suitable connection schemes. One hundred microseconds is perhaps the upper limit of delays that can be comparable to the delays involved in procedure calls.

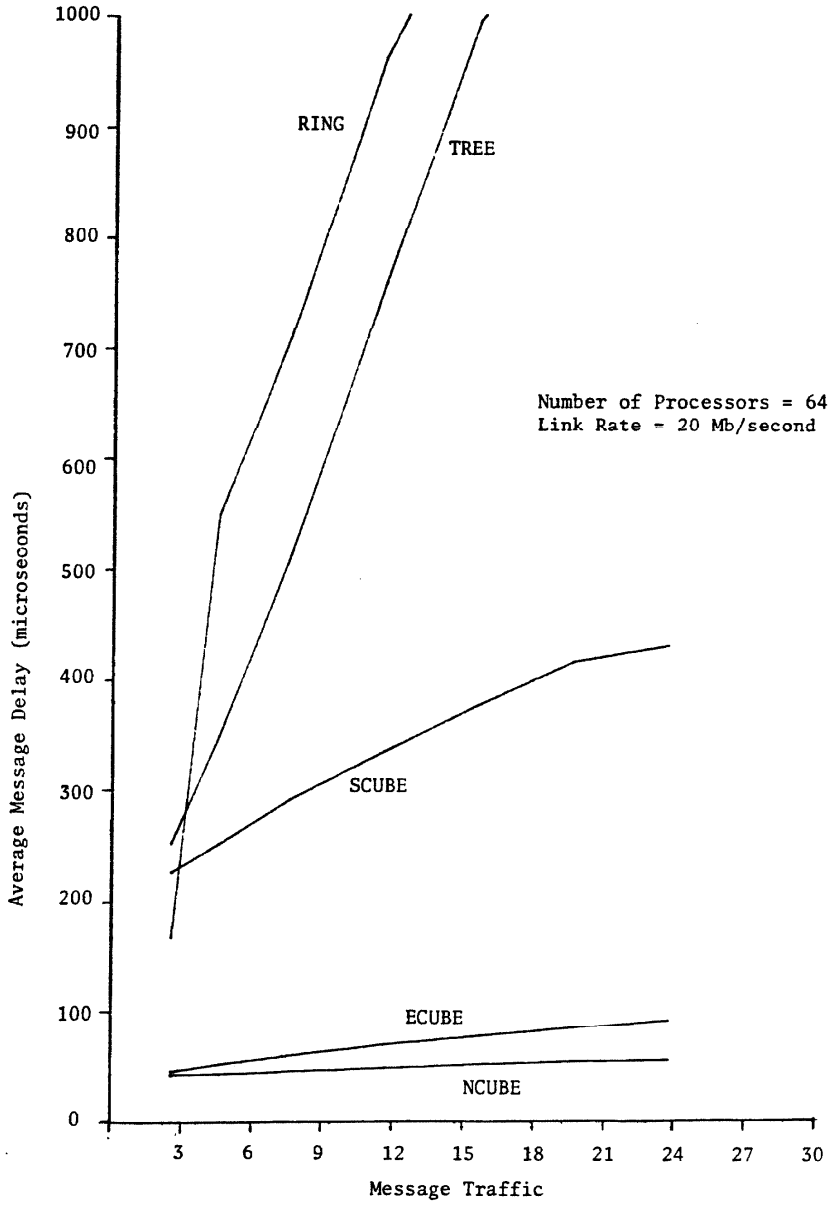


Figure 4-7

Message Delay vs. Message Locality

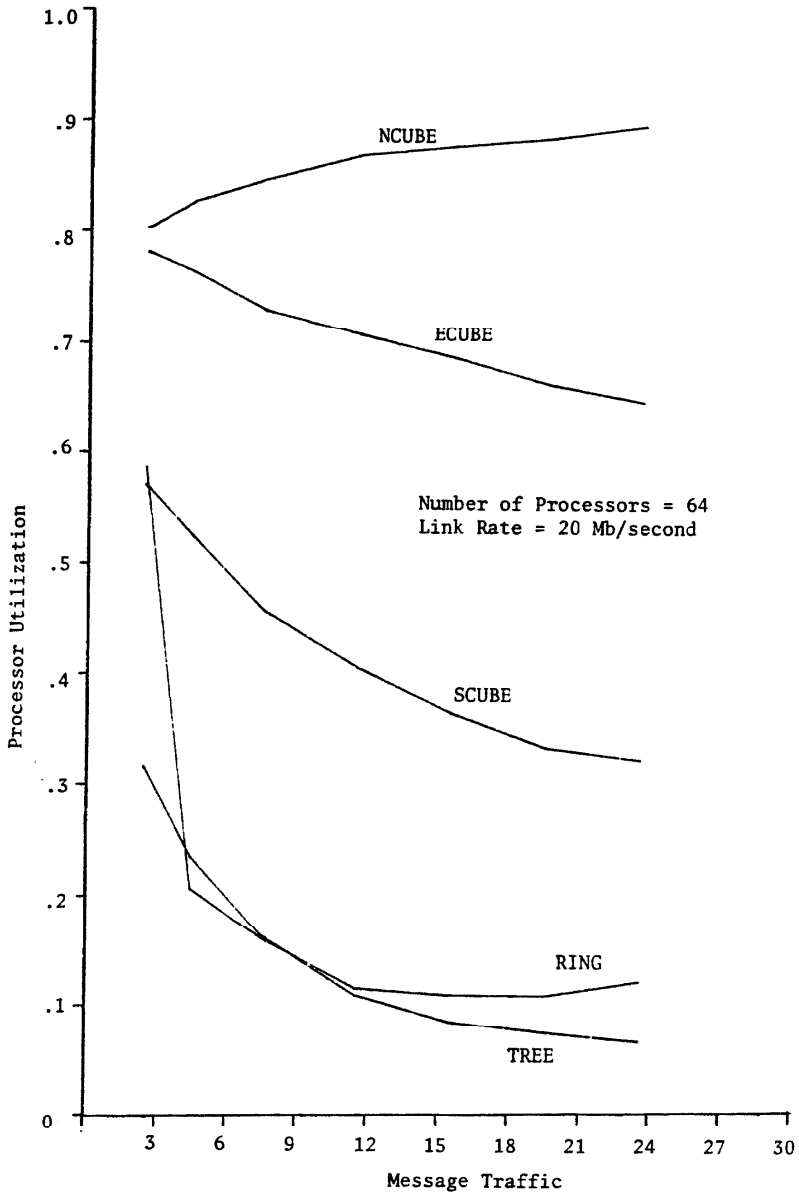


Figure 4-8

Processor Utilization vs. Message Locality

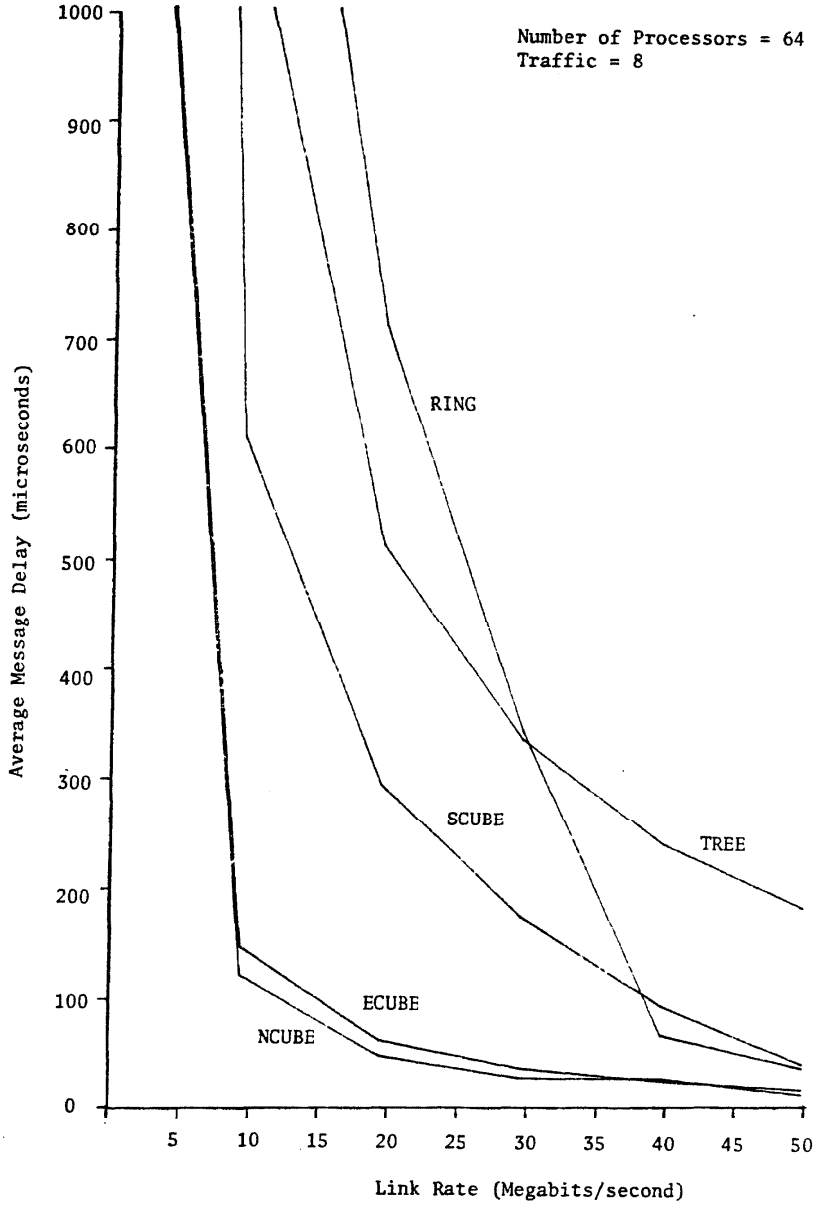


Figure 4-9

Message Delay vs. Link Data Rate

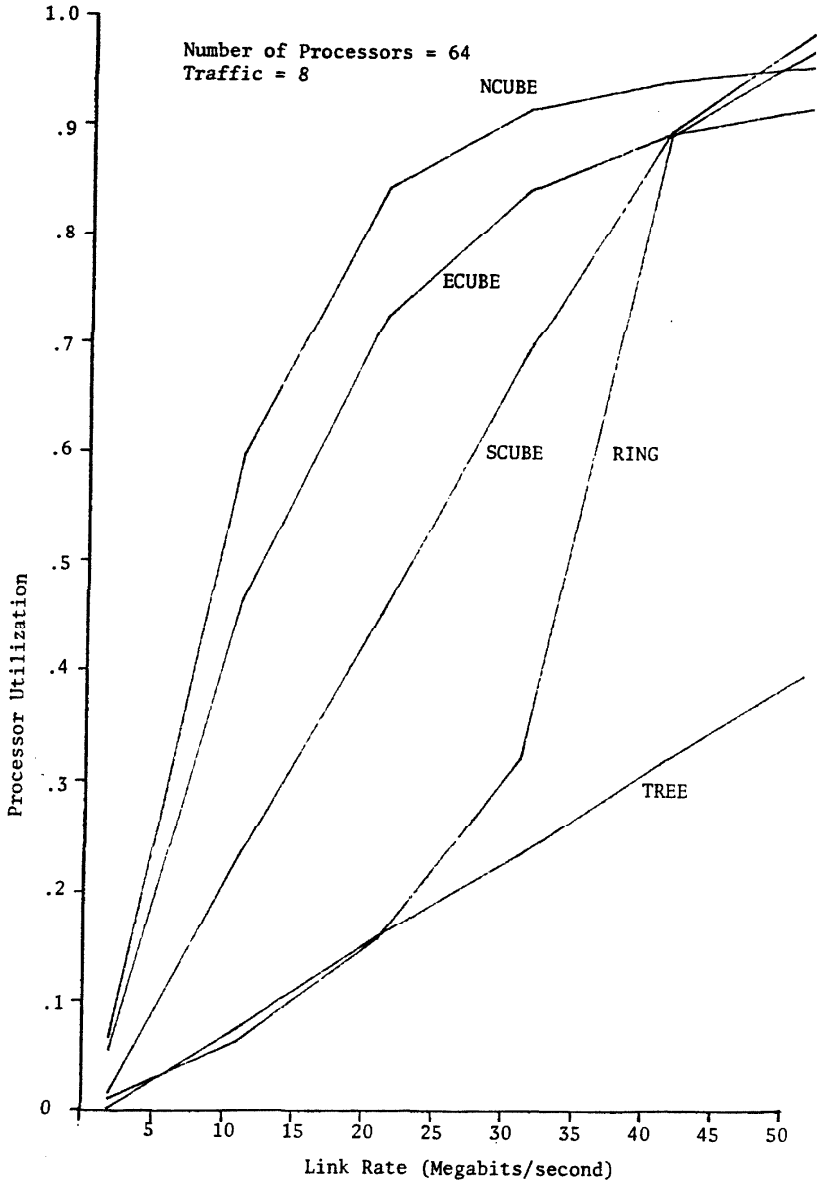


Figure 4-10

Processor Utilization vs. Link Data Rate

4.5.1. Deadlock

Deadlock was detected in several cases of simulation. This condition was detected when most or all of the queues in the simulated system filled up and packets ceased to move. The toroidal mesh was observed to deadlock consistently for message localities of 5 or greater. For this reason, the toroidal mesh is missing in most of the graphs. No attempt was made in the simulation of the toroidal mesh or the chordal ring to find methods of avoiding deadlock, though such methods may exist. To increase the probability of deadlock, a special case traffic load was simulated. In these simulations, the messages have an equal or uniform probability of being sent to any node in the network. This traffic did cause the NCUBE to deadlock for network sizes of 48, 80 and 96 nodes. This condition is indicated by dotted lines on some graphs. The ECUBE, SCUBE and TREE never exhibited any propensity for deadlock, thus supporting the contention that they are deadlock free.

The RING also never showed signs of deadlock. This is explained by the relatively large loops found in the chordal ring connection. The more links that constitute a loop, the lower the probability that they will all fill up. The ease with which the toroidal mesh, with its small loops of 4 links, became deadlocked bears out this contention. The NCUBE connection also has loops as small as 4, but with its greater number of paths, it is less likely to fill its queues.

4.5.2. Scaling of Communication Capabilities

One of the most important characteristics of interconnections strategies is their performance as the size of the network is increased. A topology which cannot maintain an acceptable level of performance for large

numbers of nodes cannot be considered suitable for application in an object-oriented machine.

In Figures 4-11 and 4-12 the characteristics of the networks are shown as a function of the size of the network. In these figures, the message traffic is highly localized with α=3. The various forms of the Boolean N-cube all improve as the number of nodes increase, due the the increasing degree of connection. The TREE connection, with a branching ratio of 4, has a nearly constant level of performance. With traffic of 3, most of the messages can be routed in two links in the TREE regardless of its size. The ARRAY and RING connections lose performance steadily as the size of the system increases, making them unsuitable. The same general behavior can be seen for other values of α in Appendix A. Figures 4-13 and 4-14 show more pronounced effects of scaling for α=8. The results of uniform traffic are shown in Figures 4-5 and 4-16.

Figure 4-17 shows how messages traveling through only one link are affected by increasing system size. Here it is observed, that the RING and, to some extent, the TREE connection exhibit increased delay in local messages as the system grows. Again, the variations of the Boolean N-cube show increasing performance as the system is made larger.

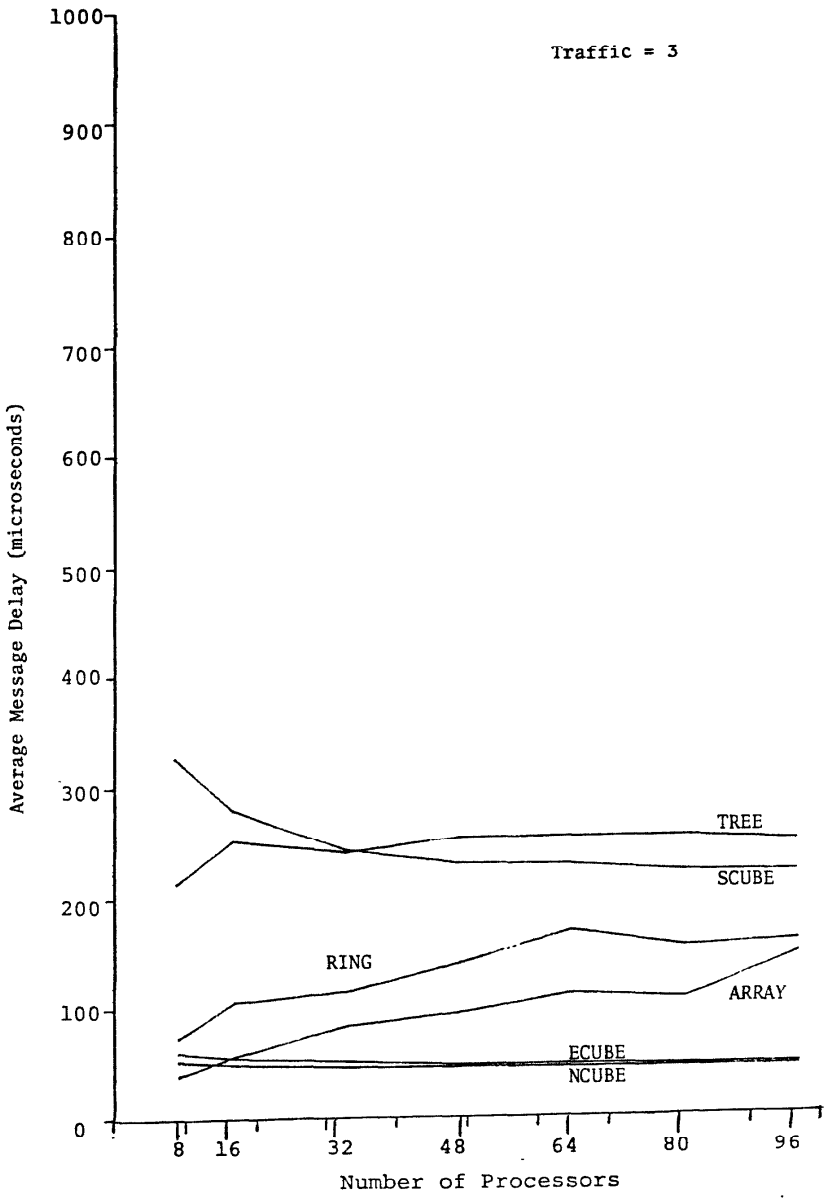


Figure 4-11

Message Delay vs. Network Size (a=3)

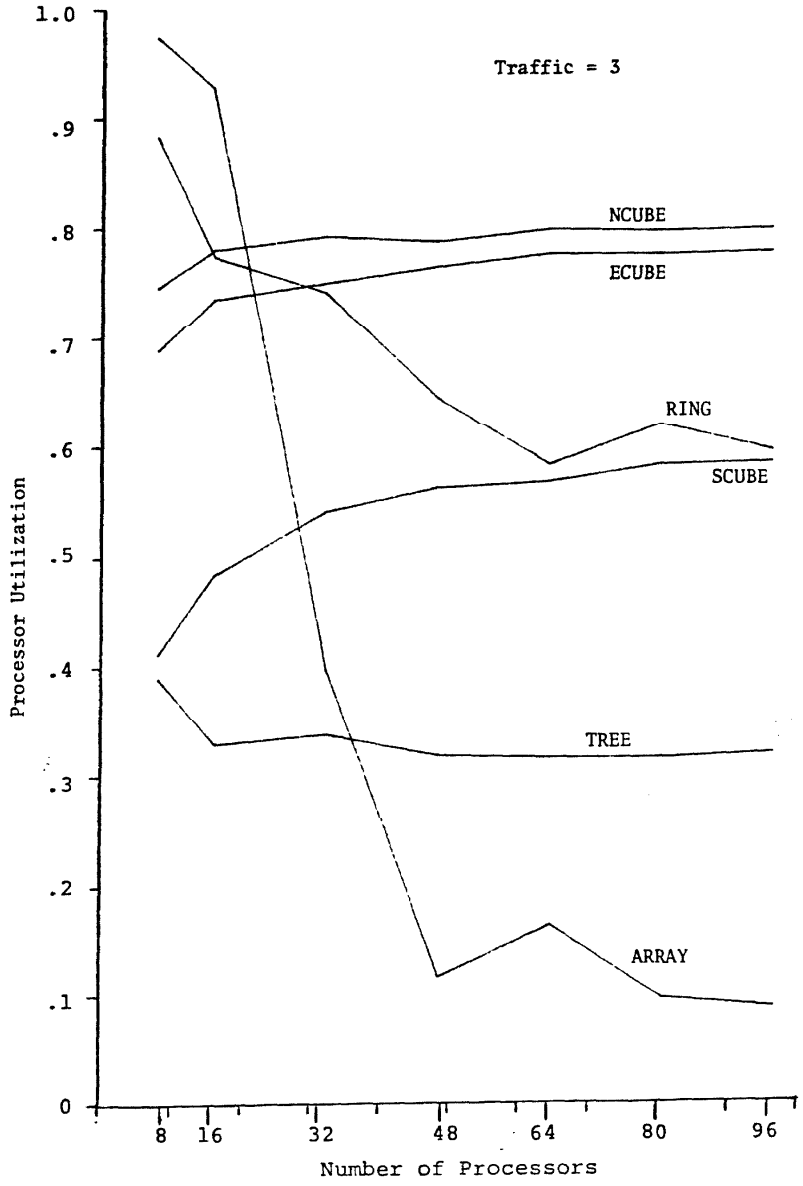


Figure 4-12

Processor Utilization vs. Network Size (a=3)

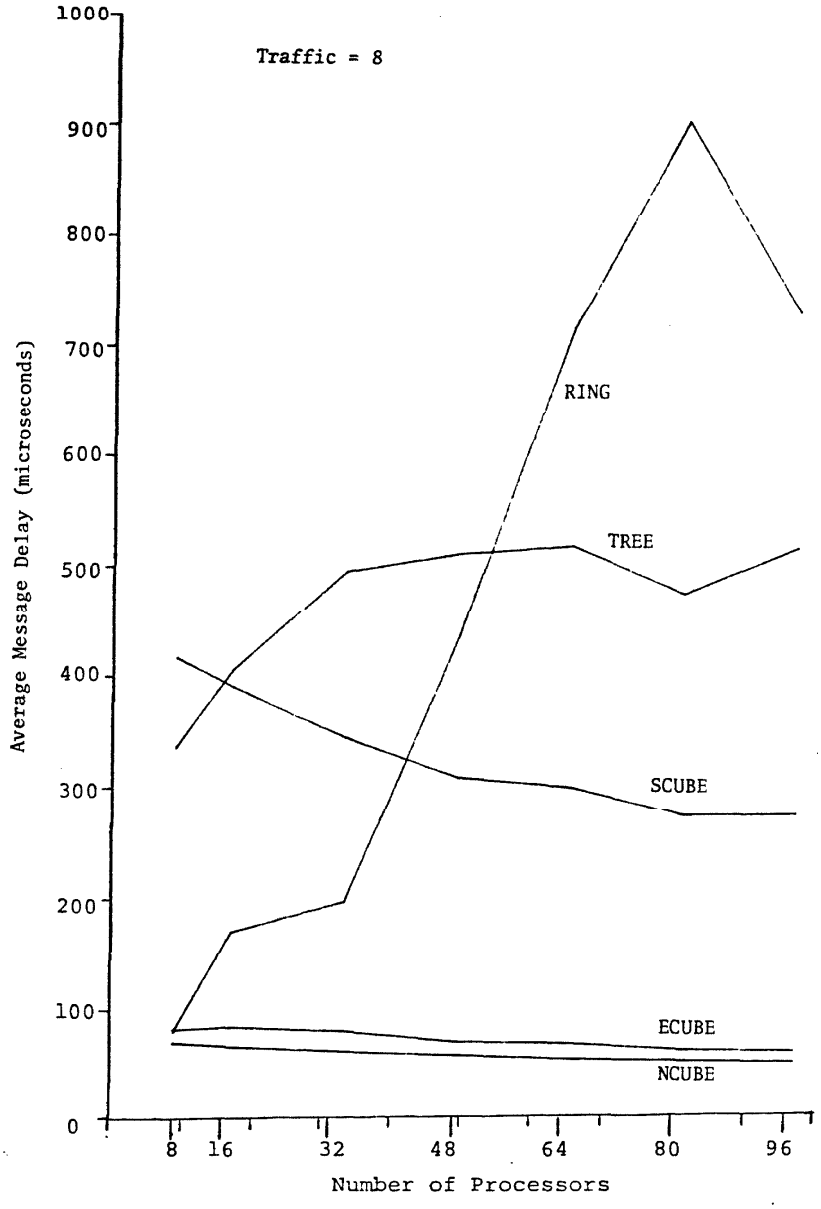


Figure 4-13

Message Delay vs. Network Size (a=8)

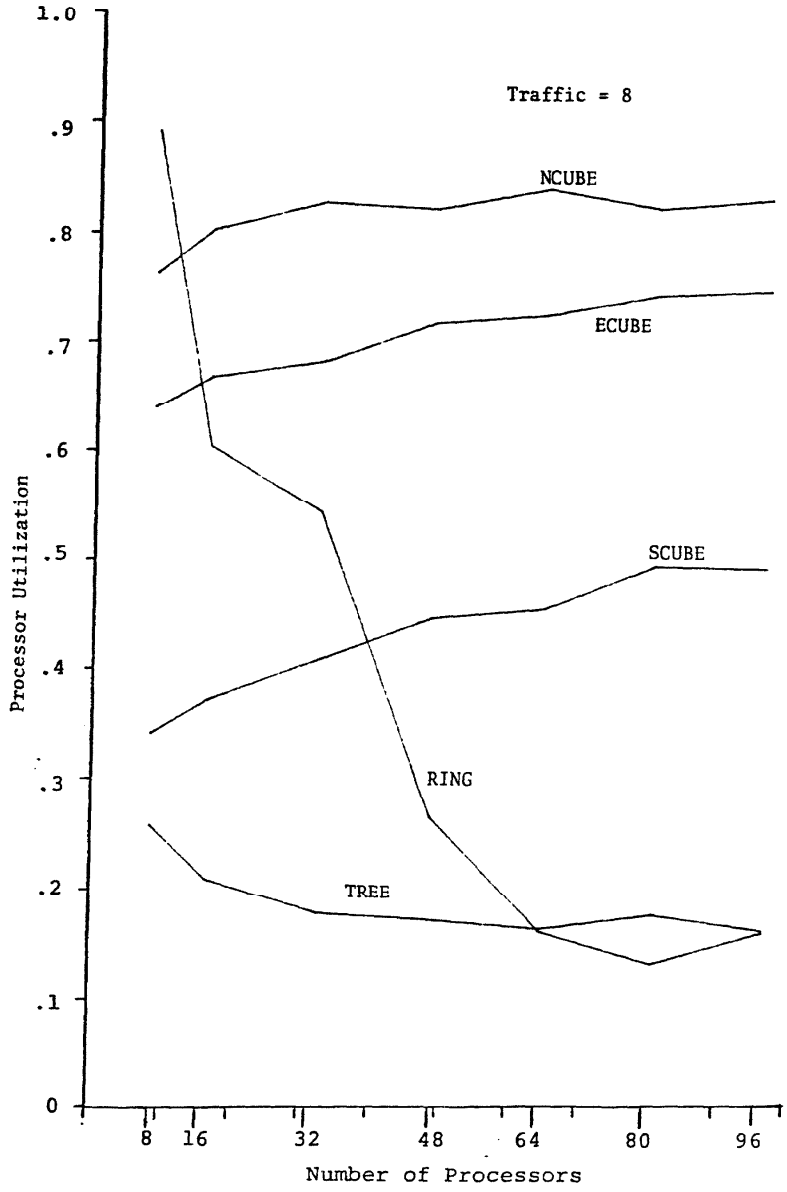


Figure 4-14

Processor Utilization vs. Network Size (a=8)

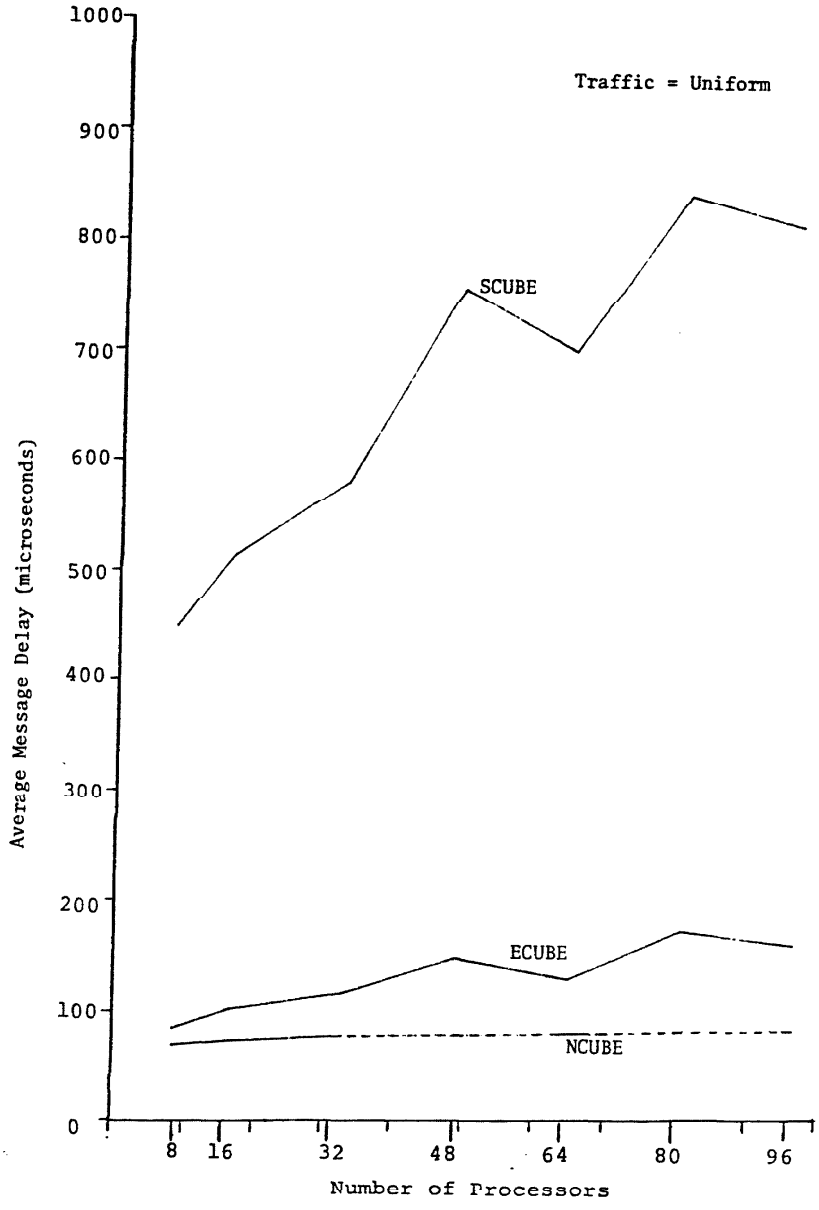


Figure 4-15

Message Delay vs. Network Size (Uniform Traffic)

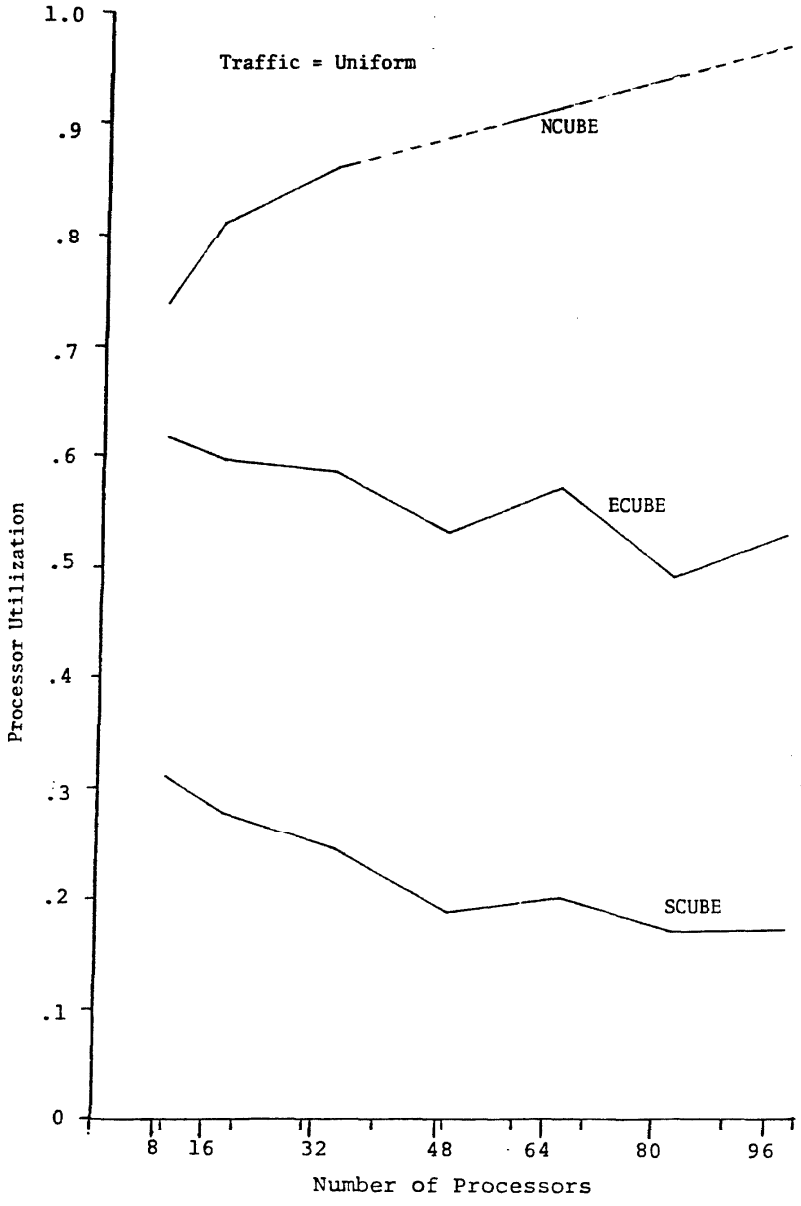


Figure 4-16

Processor Utilization vs. Network Size (Unif. Traf)

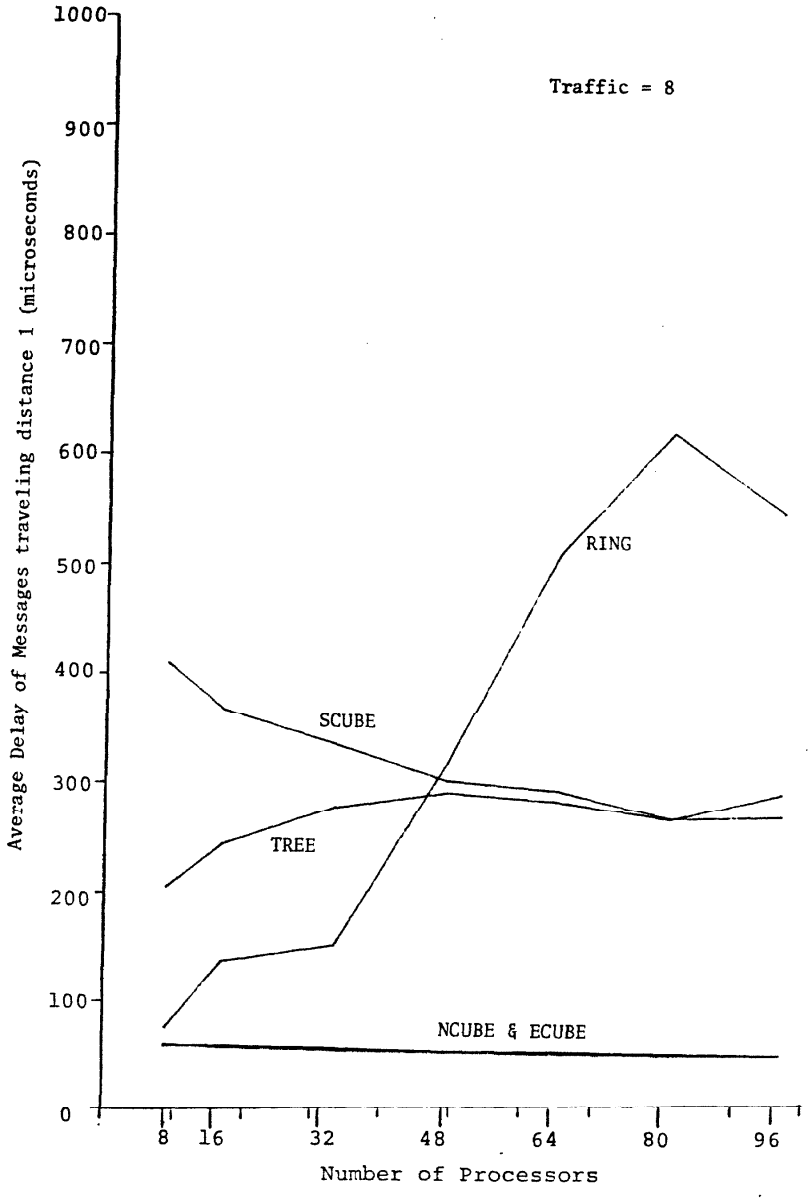


Figure 4-17

Local Message Delay vs. Network Size (a=8)

The simulation results clearly show, that of the topologies tested, only the various forms of the Boolean N-cube maintain acceptable levels of performance as the size of the system grows. As a Boolean N-cube is made larger, more communication links must be added than nodes, increasing the performance of the structure. The ECUBE version of the Boolean N-cube is observed to be only slightly less powerful than the NCUBE. The SCUBE performance scales well with system size but is considerably worse than the ECUBE at all points.

4.6. Wireability of the Boolean N-cube

The Boolean N-cube has been demonstrated to have many desirable communication properties. It is not without its pitfalls, though. There are two areas of concern about the Boolean N-cube.

This structure is inherently of a variable degree. That is, the number of connections that must be made to each node in the network is a function of the number of nodes. The ring, tree and mesh topologies are of fixed degree having a constant number of connections for any network size. Clearly, the increasing degree of the Boolean N-cube is partially responsible for its desirable communication characteristics. The number of connections required at each node in a Boolean N-cube is $\log_2 N$, where N is the number of nodes in the network. This relationship does not require that the nodes be individually modified as nodes are added to the network. A sufficient number of connections can be provided at each node in advance of their need, such that a large network can be built incrementally. If 16 connections can be provided for, a network of 64K nodes can be built. Twenty connections could as well be provided, but it is likely that other factors, such as, heat dissipation, power distribution, space requirements and wireability will limit

the size of a system to less than one million processors.

The total number of connections in a Boolean N-cube connection is $\frac{N}{2}\log_2 N$. The wiring required by this topology thus increases more rapidly than linearly with the number of processors. For any given interconnection technology, there must then be a maximum practical limit to the size of a Boolean N-cube as the costs of wiring each additional node increase. In this section, it is shown that given present packaging and wiring technology, a system as large as 64K processing nodes could be implemented. It appears that this number is presently the practical limit, though may not remain so with time. A system made up of this many processing nodes is clearly of a significant size and quite beyond the processing capability and cost of existing systems.

Given a sufficient level of integration, a processing node with 16 connections could fit in a 24 pin package. In this case half duplex communication is assumed, where only one conductor is required per link. Sixteen communication connections, four for the garbage collection algorithm and four connections for power and ground permit a standard 24 pin ceramic DIP to be utilized. This package can be mounted on a 0.8 inch by 1.4 inch grid. A 64K node network can be made up of such parts interconnected by a hierarchy of printed circuit boards and backplanes.

The processing node chips are first mounted to a conventional printed circuit board. A 4 by 32 array of these chips on a board would have a minimum size of about 7 inches by 27 inches. The interconnection topology requires that 9 connections leave the board for each processing node or 1152 signal wires. If board edge connectors are used with 10 connections per inch, about 120 inches of connector are required. If both long edges of the board

and both sides of the board are used, the long dimension of the board must be at least 30 inches, allowing for power and ground connections. Using both edges of the board requires the use of zero insertion force connectors like those used in the Symbol machine [Cowart71]. These are cam operated connectors and are somewhat exotic but workable. A simple analysis of the wiring within a board reveals that 4 signal layers are sufficient to interconnect the 128 chips and the edge connectors.

The boards can be mounted between a pair of backplanes. Half inch spacing of the boards permits 64 boards to be mounted between a pair of 30 inch by 32 inch backplanes. This 64 board assembly of the packaging hierarchy contains 8192 processing nodes. The interconnection of boards to each other within this unit could be made by wire-wrap on the two backplanes.

To complete the system, 8 such units must be interconnected. Each of the 8192 processing nodes within a unit has 3 external connections, requiring each unit to provide three sets of 8192 connections to other units. In addition to the local connections, the pair of backplanes must provide for the routing of these external connections. If two of the 3 sets of 8192 connections are on one of the two backplanes, then 16384 parallel wires must be provided for. As these wires are of substantial length, they must be near a ground plane to provide noise immunity and consistent impedance. Printed circuit boards can permit 50 etched wires per inch. Thus, about 327 inches are required. For a 30 inch wide backplane, 12 layers are needed. A backplane with these features would be unusual and not inexpensive but it is well within the capabilities of current technology. Eighteen layer printed circuit boards are used routinely in military and aerospace applications.

A method of organizing the 8 units required to complete the system is shown in Figure 4-18. The dimensions of the backpanel pairs (about 30 by 32 by 10 inches) make the overall size of the system, less the power supplies, about 8 feet long by 4 feet high by 3 feet deep. The connections shown between the backplanes as bars in the drawing each represent 8192 wires. These connections are difficult to provide. They could be made using a flexible printed circuit board with a 30 inch by 3 inch block of wire wrap pin connectors on each end for connection to the backplanes. Flexible printed circuit technology or large quantities of ribbon cable could also be used. The longest of these connections would be approximately 8 feet long.

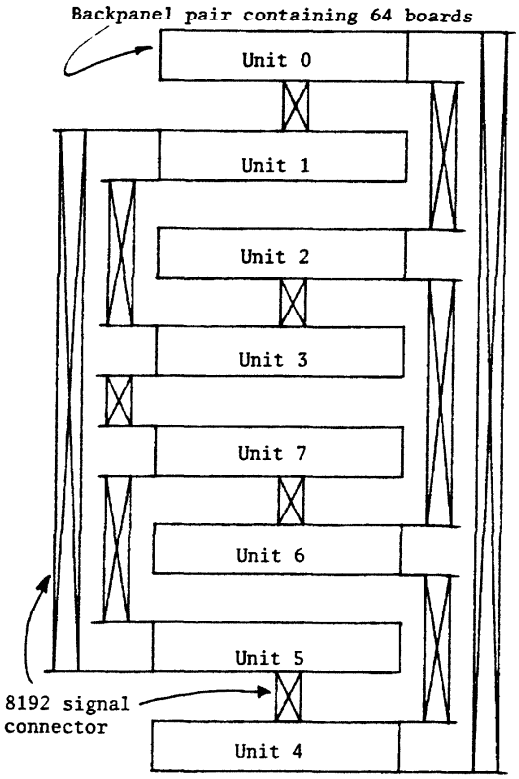


Figure 4-18

Top Level Interconnection of 64K Node Boolean N-cube

Power distribution can be accomplished at the edges of the backplanes. The heat dissipation of such system could present a problem. If each processing node dissipates less than 0.25 watts, the system could be air cooled, though with difficulty. Liquid cooling might very well be needed if the processing nodes are not implemented in a low power technology such as CMOS.

With existing, though somewhat exotic, packaging techniques 64K processing nodes can be connected in a Boolean N-cube. This size of system is sufficient to be considered interesting and could yet grow with advances in packaging and interconnection technologies.

4.7. Conclusions

It is the purpose of this chapter to show the existence of a suitable topology for interconnecting large numbers of processing nodes to form an object-oriented machine. Of the classes of structures investigated, only the Boolean N-cube exhibits the performance and scaling characteristics required. Owing to the possibility of deadlock, traffic routing must be modified to prevent deadlock from occurring. A message routing scheme was presented which is deadlock free. A non-queued communication system was presented which is also deadlock free and which scales well with system size. However, the overall performance of this scheme was considerably lower than that of the queued Boolean N-cube with restricted message routing. This queued Boolean N-cube, using Sullivan's routing algorithm is then the best choice, as it fulfills all of the requirements stated.

The choice of the Boolean N-cube as a suitable interconnection topology does not imply that it is the only suitable topology. Other schemes may work as well and have lower implementation costs. Some variation of the

hypercube structures proposed by [Wittie81] show some promise of this. The relatively low duty cycle exhibited by the links in the Boolean N-cube suggests that contention networks with a limited number of nodes per bus might be used rather than point-to-point connections. It seems clear from the simulation results presented here that networks of low degree generally lack the performance and scaling necessary to this application.

Chapter 5

A Localized, Virtual Object Environment

5.1. Introduction

The object-oriented, homogeneous machine presented in the previous chapters, raises some new problems. This chapter proposes potential solutions to some of these problems. However, these solutions are unsupported by analysis or simulation but provide a starting point from which a working system can be designed. The methods presented here for controlling the locality of reference and the locating of objects are heuristic and do not lend themselves well to *a priori* study. A convincing analysis of them will require a better understanding of concurrent object oriented programming.

In the architecture described, objects must move within the structure to provide real concurrency, to balance the processor and memory loads and to maintain some degree of locality with each other. The memories of the processing nodes must not directly limit the number of objects that can exist in the system at one time. Mass storage devices must be used to hold inactive objects until they are referenced. As one object sends a message to another object, it must be possible to determine the processor in which the destination object is located with some ease.

This chapter suggests solutions to these problems. In all cases the solutions are of a heuristic nature and are greatly affected by the machine

structure, the processing nodes, and the programs they execute. This characteristic makes performance predictions difficult and unreliable. For any particular set of heuristic procedures it is possible to contrive situations in which they perform poorly. On the other hand, for any situation one can usually contrive heuristics that will have the desired effect. The algorithms presented here are analogous to the scheduler and swapping strategy in a conventional system of today, in that, they will require "tuning" to achieve a desired level of performance.

Three areas of concern are addressed. First, a strategy for moving objects between processing nodes to preserve locality and concurrency is presented. Related to this strategy is a means of providing an virtual object environment, one in which an object need not be resident in a processing node memory but may be stored on a device such as a disk until it is referenced. A method for finding the processor in which a given object resides is also presented. As with the garbage collection algorithm, these techniques must not require global communication but must use local data to insure that they scale well as the size of the system is increased. The system is assumed to have a Boolean N-cube topology as suggested in Chapter 4.

5.2. Maintaining Locality Among Object References

The most immediate need for moving objects from one processor to another occurs when objects are being created in one processor and it exhausts its heap area. We assume here that objects are always created in the same processing node as their creator but may be subsequently moved. When a processing node runs out of some resource, such as memory, it must be able to send excess objects to another processor. Some of the objects

may be garbage but cannot be presumed garbage until the garbage collection algorithm completes its current cycle.

If objects cannot be moved between processors, there can be little opportunity for concurrency, since the objects will have to share a single processor. A message from an object that requires no response cannot cause real concurrent behavior unless the source and destination objects are in different processors. If, on the other hand, two objects consistently communicate with messages requiring a response, then it is desirable to place the objects in the same processor to minimize communication costs.

The motion of other objects and reference variables in the system may result in excessive communication costs between objects. If several closely related objects reside in distant processing nodes, there will be a longer communication delay between them and more message traffic generated than if the objects were resident in neighboring processors.

In data flow programs the objects or operators of the program are fixed and communicate with each other by a fixed topology. For such systems, a resource assignment can be worked out in advance of the program execution to assign the operators to various processors in such a way as to minimize the cost of communication in the program. The work of [Wu80] presents an algorithm for this purpose.

In the object-oriented environment, the topology of the communication changes as new objects are created, as old objects are removed and as reference pointers are exchanged between objects. This dynamic behavior requires a dynamic, run-time means of preserving some level of locality between related objects. To make decisions concerning which objects to move and where to move them, some measure of communication costs must

be accumulated for each object. Storing and updating such data will, of course, consume resources.

Each object can have associated with it a quantity for each port of the processing node. Each port can be thought of as a direction in N-space. If the quantity is constantly updated to reflect the amount of message traffic to the object in each direction, a determination can be made of whether the object should be moved. If one or more of the directional costs becomes sufficiently larger than the others, it is an indication that a closely related object resides in the associated direction. The object can then be moved to the neighboring node in that direction, placing it closer to the source of the message traffic.

Many heuristic schemes can be developed to make selections based on such data. Here we present a simple scheme using several controlling constants to regulate the policy of the decisions. With each object there is a list of $logN+1$ variables. The additional quantity is associated with traffic within the object's processor, the other quantities are each associated with a communication port.

$x_i = x_0, x_1, x_2, x_3 \cdots x_d$

where $d = \log_2(number\ of\ processors)$

As the machine runs, the quantities are updated in the following manner. This task could be done by the communication processor or by special purpose hardware to avoid unnecessary load on the the object processor.

For a Message on Port j

$$x_j \leftarrow x_j + \frac{MsgLength}{Rate}$$

Rate is a constant controlling how quickly x_j responds to message traffic. A large value will reduce the effect recent traffic might have on the value of x_j.

As execution proceeds, x_j will rise with message traffic. If resources are needed, or other factors warrant it, the objects may be tested in the following manner to find candidates to be moved. A threshold value T is first computed.

$$T = \frac{Resist}{d} \sum_{i=0}^{d} x_i$$

Resist is a constant controlling how strongly objects resist motion. *Resist* would usually be greater than 1, with larger values causing objects to move less frequently for identical message traffic. The quantity T is a threshold against which the individual x_j are tested. If any x_j exceeds T then the object will be moved. The largest x_j selects the direction in which the object is to be moved. The object is sent to the neighboring node connected to the port associated with the x_j and when it arrives all its x_j are set to zero.

To accommodate the needs of concurrency, a modification of this procedure will push concurrent objects away from each other and attract non-concurrent objects. If x_0 is incremented only when the message does not require a response, then x_0 will become a measure of lost opportunities for concurrency. Thus, if x_0 both exceeds the threshold and is the largest x_j then it indicates the object should be moved out of the processor it now resides in and moved to one close by to take advantage of concurrency. If x_1 is incremented only for messages that require a response, then concurrently executing objects will not generally be moved into the same processor.

To prevent instabilities in the system, such as, objects chasing each other or oscillating between processors, they must be given inertia. A simple

time-of-day stamp on the object, made each time the object is moved, would prevent the object from again being moved until a specific period of time had elapsed. The period of time could be computed based on past behavior or could be a constant. Too rapid movement of objects would consume excessive communication bandwidth and adversely affect the ability of the system to locate specific objects.

Variations in the controlling constants and in the details of the procedure permit it to be adapted to a range of object environments. Before machines of the type described here are built and programmed, it will be very difficult to predict the effectiveness of heuristic methods such as these. However, it is clear that this method is one that could dynamically preserve the locality of reference in such a system. Since it makes its decisions on purely local data it will scale well in progressively larger machines.

5.3. Providing a Virtual Object Space

As with conventional machines, the size of a program that can be accommodated by the system should not depend on the number of processors or on how much physical memory they have. Paging and segmentation techniques have evolved in von Neumann architectures to make the memory address space available to user programs independent of real memory constraints. The basic addressable unit of an object-oriented machine is the object and this section presents a method for permitting a machine to manipulate and execute more objects than it has physical memory to store at one time.

Virtual memory, and hence virtual objects, are not really virtual at all but are quite real. All such schemes require additional memory for those parts of a program that do not fit into the machine's real memory. Usually

this additional memory is a mass storage device such as a disk. Virtual memory systems must allocate sufficient swapping space on a disk to store the address spaces of some maximum number of processes. In a virtual object environment, the size of the available mass storage devices will determine the maximum number of objects that can be supported in the system.

To prevent bottlenecks from being formed in the system, it is clear that mass storage devices should be distributed among the processing nodes rather than be concentrated at one location. The disks must be assigned to processing nodes in such a way that all processing nodes can conveniently communicate with the disk closest to them. In the Boolean N-cube this effect can be had by assigning a disk to every processing node with a 0 *mod* 4 or 0 *mod* 8 address. This assignment would place one disk in the system for every 4 or 8 nodes. The ratio can be changed to distribute the desired amount of storage throughout the machine. It is important that the ratio of the number nodes to the number of disks be a power of 2 such that a processing node need only clear the lower n bits of its own address to generate the address of the nearest node with a disk, where 2^n is the ratio.

During the execution of program objects, a node will be able to periodically recover memory occupied by garbage but will also be called on to create new objects and accept objects from other nodes. When the node's memory resources become diminished, it will have to select objects to be moved to free up the memory and processing resources they consume. The preceding section showed how objects can be selected to be moved for execution in other nodes with the intention of reducing communication delays and costs. Additionally, objects should be moved to the disk if they

become inactive.

An inactive object is one that either has no messages to process, or one that is waiting for a response from another object. If an object remains in one of the above states for an extended period, it is clearly a good candidate for removal to the disk. This condition is analogous to a page in a virtual memory system that has not been recently referenced and is then swapped out of memory to the disk to make room for another page. An object that has no messages to process may be garbage as well as inactive, if it is garbage it will be eliminated at the end of the next garbage collection cycle. An object that is waiting for a response from another object is clearly not garbage but can nevertheless be swapped to disk until the response is available.

Objects can be time stamped when they send or receive messages or engage in other computational activity. When the processor scans its objects for candidates for removal, one criterion will be the length of the interval since each object was last active. Objects can first be moved based on their communication with other objects as suggested in the previous section. If the node must eliminate more objects then it must select those that have been inactive longest and move them to the closest node with a disk. The time stamp allows the nodes to implement a least-recently-used algorithm on inactive objects. The transfer of objects in this manner will cause them to be considered to reside in the node with the disk. If messages are received for these objects they will then resume their activity and perhaps be moved to another processing node.

A processing node with a disk can be expected to receive a steady stream of incoming inactive objects. Since all garbage objects will also be

inactive objects, all garbage will eventually be sent to these nodes unless they are collected first. Also, as these nodes receive messages for their objects, there will be a continual stream of re-activated objects moving away from the nodes with disks. The difference in the volume of the two streams will be the garbage objects left in the node and never referenced. The nodes with disks must implement the garbage collection algorithm just as any other node must but they must include all the nodes on their disks in the algorithm.

The special responsibilities of the nodes with disks require additional hardware support. To operate the disk drive a specialized processor can made part of the node communicating with the object memory. The scanning of the disk in performance of the garbage collection algorithm can be delegated partly to this processor. To maintain a level of performance consistent with the other nodes in the system, the nodes with disks will require space for a larger table of objects to accommodate those objects on the disk. The object tables that must exist in each node are further discussed in the next section. To provide space for the table additional memory must be available to nodes with disks. If the technology permits, an additional level in the storage hierarchy could be added to these nodes in the form of magnetic bubbles or CCDs. The decreased access time of these devices would improve the response time of these nodes.

To some extent, the addition of disks and their accouterments to the system will destroy some of the homogeneity of the machine. However, mass storage devices of some kind are a necessary ingredient to any real system and must be included. Homogeneity is preserved if one considers that its granularity has been increased so that the basic unit of replication in the

system is a set of 4 or 8 processing nodes, one of which includes a disk drive.

5.4. Locating Objects in the Network

One of the fundamental services the run-time systems executing in the nodes must perform is the ability for one object to send a message to another object regardless of what node it may reside in. The motion of objects, whether for preservation of locality or for inactivity, will make an object a moving target with respect to messages intended for it. Clearly, the nodes do not have the memory resources to store a table entry locating each object in the system, nor could such tables be kept consistent as objects are moved. In this section, the problem of maintaining a small table and sending messages to the proper processor are addressed.

To begin, each node will maintain a table of a limited size. The table will have to contain an entry for every object actually resident in the node, therefore the size of the table will be related to the amount of real memory available to the node. The table will contain entries for other objects as well. Object identifiers will, no doubt, be large integers of perhaps 32 or more bits. To efficiently access the table with object identifiers, the table will have to be hash coded.

When a message is received by a node, or when an object in the node attempts to send a message, the table entry for the destination object is accessed. If an entry exists, it will indicate the address of the node in which the object was last know to reside. For incoming messages, the destination node should be the address of the current node, if so, the message is given to the indicated object. If the entry indicates some other node, then the object has been moved and the entry was updated to point at the object's new home. The message is then forwarded to the indicated node and a message

is sent to the originator indicating the new node in which the object can be found so that it can update its own table entry for that object.

For outgoing messages, the message is sent to the node indicated by the table entry. The entry may refer to the same node and the message is directed at another object in the same processor. Otherwise, a message is sent to the node indicated by the table entry. If the destination node has no entry for the destination object, it sends the message back to the originator indicating the object is unknown.

When messages are returned because the destination object was unknown to the destination node, or when there is no table entry for the destination object of an outgoing message, the processing node must determine where in the system the object resides. To determine the address of an object, every node in the system must be asked if it "owns" the object or objects in question. Sullivan [Sullivan77] presents an algorithm whereby a message can be broadcast to all nodes in a Boolean N-cube with no redundancy. The time required to do so is $\log_2 N$. Using a broadcast message, every node can be asked to respond to the originator of the message if it owns any specified objects. Processors with disks must respond even if the object is on their disk, thus such nodes will require correspondingly larger tables. The response is used by the originator to construct a new table entry for that object.

Since the tables are of a limited size and clearly cannot hold entries for all the objects in the system, there must be a means of freeing space in the table for new entries. Table entries, like objects, can be time stamped to indicate the time of their last access. When additional space is required in the table, old entries are destroyed on a least-recently-used basis. Entries

for objects owned by the processor itself can never be purged until the object is moved to another processing node.

The broadcast message used locate individual objects in the system is global communication, and as such, cannot be used except infrequently. To insure that probability of not finding a table entry is low, the tables have to be large enough to hold a "working set" set of entries. The concept of a working set of objects brings the issue back to one of locality. If those objects that communicate frequently amongst each other do not reside in neighboring processors, the number of entries that must be stored in the object table will have to be large. Also, if objects are permitted to move between nodes too quickly, the forwarding of message will slow down communication and increase the number of table entries used in the system. A real, working system running substantial programs will be required to find the best tradeoffs between these issues and to find a suitable range of parameters for the object tables and the object transfer policies.

Chapter 6

Conclusions and Summary

In this thesis, the essential elements of a general purpose, homogeneous machine have been presented. To provide a good fit with VLSI technology, the machine would consist of a potentially large set of regularly connected processors, each with their own memory. While many such machines have been proposed, none can be considered general purpose by the definition of Chapter 1. To provide a general purpose programming environment, a modification to object oriented languages such as Simula was shown to provide a convenient notation for concurrency as well as the modularity, locality and compartmentalization necessary in the the system.

Object oriented languages provide a natural programming concept which encourages the user to arrange programs as data objects which are defined to include the operations related to them. Objects communicate with other objects by passing messages. Messages are queued for each object, where various objects may execute the operations required by the messages concurrently. Concurrency has been made available to the programmer by making a small extension to the semantics of message passing. The simplicity of using object oriented programming concurrently not only makes concurrent programming more convenient but makes the specification of deterministic programs less error prone. The utility of this style of concurrent programming was illustrated with several practical programming problems.

One important characteristic of a homogeneous machine must be its ability to provide increased performance in proportion to its size. To meet this requirement neither the hardware not the software can employ techniques that degrade as the system grows. To permit performance to scale upward, we have shown that the interconnection of the parts of the machine must be of a higher degree than tree, ring or mesh connections provide. Specifically, the Boolean N-cube connection has been shown to provide the necessary performance for various sizes of systems.

In providing an object oriented environment in a homogeneous machine, several new problems arise that are not found in conventional von Neumann architectures. The first of these is distributed, on-the-fly garbage collection. A new algorithm was developed which meets the needs of a homogeneous machine. Its performance scales well for increasing numbers of elements in the system. The algorithm is simple and sufficiently general purpose such that it may find usage in other applications, such as distributed data base systems. The garbage collection algorithm does not depend on shared variables or reference counts or on intricate pointer structures. It accomplishes its task with a minimum of overhead costs. Simulation of the algorithm has demonstrated that it performs adequately.

Locality must be maintained in a homogeneous architecture. Object oriented programs are dynamic, where objects are created and destroyed at a high rate and where object pointers change their topology rapidly. In a dynamic environment, the objects must be enabled to move about in the system, both to insure concurrent execution and to minimize communication delays. A heuristic method was presented that preserves the locality of references between objects in the system. Objects that can

execute concurrently are moved to separate processing nodes so that they may do so. Objects that communicate heavily are moved to neighboring processing nodes to reduce communication delays. Methods for locating objects in the system and for moving less active objects to mass storage devices have also been presented.

The sum of these techniques is a programming environment that supports concurrent, object oriented programming on an ensemble of identical processors. This system has the ability to provide greater performance by the addition of more processors. This ability is seen only to a small extent in existing machines, for which it may be possible to add only two or three more processors to a system. This homogeneous architecture will scale well for thousands of processors. The advancement of general purpose programming to large numbers of processors requires the algorithms and techniques described in this thesis. A suitable concurrent programming language, a suitable interconnection topology, as well as garbage collection and localization procedures which scale with system size, have all been addressed and solved herein. In effect, we have shown how a large number of interconnected processors can be programmed in a simple style to work together on the same problem without regard to the size of the system or the exact nature of its components.

A full scale implementation of the system presented in this thesis will, no doubt, raise new questions. An implementation will also provide the opportunity to find suitable values for the many parameters of the system. Some of the questions that must be answered are: What instruction set architecture is best suited to object oriented languages? How much memory should a processing node have? How large should mass storage devices be

and how much additional memory will they require? How best to organize an operating system and what should its functions be?

The object concept pervades the entire machine and can be expected to be felt in the services the operating system. Many of the traditional concepts of files, processes, tasks and jobs will be modified in such a system. Any object can be thought of as an independent task or process. The concept of a file as a sequence of bytes could be replaced by structures of objects where attributes implement any access method the programmer defines.

Ensemble machines, consisting of large numbers of identical parts, should be able to provide a very high degree of reliability due to the redundancy of the structure. If any single processor in a Boolean N-cube structure fails, no other node in the structure is isolated. The structure should be able to operate in a degraded configuration until such parts are repaired. A more difficult problem is to provide the necessary data redundancy and backup in the system to permit the system to continue its execution in the presence of a faulty node. As the number of nodes in the structure grows larger, reliability issues will achieve greater importance. One virtue that will remain with homogeneous machines of any size is their ability to easily isolate and repair faulty nodes. The regularity of the structure makes isolation of faults to the correct node much simpler than fault isolation in large single processor machines and the use of identical parts makes its repair trivial.

The next step to be taken in this line of research is the construction of a test vehicle. Using existing microprocessor technology with the addition of custom communication hardware, a processor/memory node can be built on

a single printed circuit card. A machine consisting of 128 or 256 such boards could be assembled in the space of several racks. Mass storage devices would not be necessary in the test vehicle but could be added in the future. A run-time system based on the techniques and algorithms of Chapters 3 and 5 must be written to be resident in every processing node. A cross compiler must be developed to compile concurrent Simula as presented in Chapter 2 for execution in the processors of the test vehicle.

A test vehicle will validate the ideas presented in this thesis and will enable suitable parameters to the heuristic algorithms of Chapter 5 to be found. Given a compiler and run time system for the test vehicle, the means to write general purpose programs for the machine will be available. As a body of experience with programming a homogeneous, concurrent machine develops, the run time system of the machine can be improved to meet requirements of real programs.

Until machines exist which provide positive incentives for concurrent programming, concurrent notations and algorithms will be restricted to research topics. With the advent of inexpensive computing structures made possible by VLSI, such machines can now be economically constructed. This thesis has shown how a system can be built around such a machine that will fulfill the basic requirements of general purpose programming. This approach is a major departure from the von Neumann style of computer architecture yet it fits well with both the integrated circuit technology and a simple concurrent programming model.

Bibliography

[Almes80]
Guy T. Almes, "Garbage Collection in an Object-Oriented System", Ph.D Thesis, Department of Computer Science, Carnegie-Mellon University, June 1980.

[Anderson75]
G. A. Anderson and E. D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics and Examples", *ACM Computing Surveys*, Volume 7, pp. 197-213, December 1975.

[Arden81]
Bruce W. Arden and Hikyu Lee, "Analysis of Chordal Ring Network", *IEEE Transactions on Computers*, Volume C-30, pp. 291-295, April 1981.

[Arnborg72]
Stefan Arnborg, "Storage Administration in a Virtual Memory Simula System", *BIT: Nordisk Tidskrift for Informationsbehandling*, Volume 12, pp.125-141, 1972.

[Arvind81]
Arvind and Vinod Kathail, "A Multiple Processor Data Flow Machine that Supports Generalized Procedures",*Proceedings of the 8th Symposium on Computer Architecture*, pp.291-302, May 1981.

[Backus78]
John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", *Communications of the ACM*, Volume 21, Number 8, pp. 613-641, August 1978.

[Baker78]
Henry Baker, "Actor Systems for Real-Time Computation", Massachusetts Institute of Technology, Laboratory for Computer Science, TR-197, March 1978.

[Benes65]
V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

[Bentley79]
J. Bentley and H. T. Kung, "A Tree Machine for Searching Problems", *Proceedings of the IEEE 1979 International Conference on Parallel Processing*, Bellaire, Michigan, August 1979.

[Berkling75]
K. J. Berkling, "Reduction Languages for Reduction Machines", *Proceedings of the 2nd Symposium on Computer Architecture*, pp. 133-140, 1975.

[Birtwhistle73]
G. M. Birtwhistle, O-J Dahl, B. Myrhaug, K. Nygaard, *Simula Begin*, Petrocelli, New York, 1973.

[Birtwhistle79]
G. M. Birtwhistle, *Discrete Event Modelling on Simula*, Macmillan, June 1979.

[Bobrow80]
D. G. Bobrow, "Managing Reentrant Structures Using Reference Counts", *ACM Transactions on Programming Languages and Systems*, Volume 2, pp.269-273, July 1980.

[Browning80]
Sally Browning, "The Tree Machine: A Highly Concurrent Computing Environment", Ph.D Dissertation, Computer Science Department, California Institute of Technology, 1980.

[Brinch Hansen75]
Per Brinch Hansen, "The Programming Language Concurrent Pascal", *IEEE Transactions on Software Engineering*, Volume 1, Number 2, pp 199-207, June 1975.

[Brinch Hansen78]
Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept", *Communications of the ACM*, Volume 21, pp.934-941, November 1978.

[Chen82]
Marina Chen, Martin Rem and Ronald Graham, "A Characterization of Deadlock Free Resource Contentions", Computer Science Department, Technical Report Number 4684, California Institute of Technology, January 1982.

[Clinger81]
William D. Clinger, "Foundations of Actor Semantics", Artificial Intelligence Laboratory, Massachusetts Institute of Technology, AI-TR-633, May 1981.

[Collins60]
George Collins, "A Method for Overlapping and Erasure of Lists", *Communications of the ACM*, Volume 3, pp.655-657, December 1960.

[Cowart71]
B. E. Cowart, R. Rice and S. F. Lundstrom, "The Physical Attributes and Testing Aspects of the SYMBOL System", *Proceeding of the AFIPS Spring Joint Computer Conference*, Volume 38, pp. 589-600, Spring 1971.

[Davis78]
A. L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine", *Proceedings of the 5th Symposium on Computer Architecture*, pp.210-215, April 1978.

[Dennis74]
Jack B. Dennis and David P. Misunas, "A Preliminary Design for a Basic

Data-Flow Processor", *Proceedings of the 2nd Symposium on Computer Architecture*, pp.126-132, December 1974.

[Despain78]
A. M. Despain and and D. A. Patterson, "X-Tree: A Tree Structured Multiprocessor Computer Architecture", *Proceedings of the 5th Symposium on Computer Architecture*, pp. 144-151, April 1978.

[Deutsch76]
L. Peter Deutsch and Daniel G. Bobrow, "An Efficient, Incremental, Automatic Garbage Collector", *Communications of the ACM*, Volume 19, pp.522-526, September 1976.

[Dijkstra78]
Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten and E. F. M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation", *Communications of the ACM*, Volume 21, pp.966-975, November 1978.

[Finkel80]
R. A. Finkel and M. H. Solomon, "Processor Interconnection Strategies", *IEEE Transactions on Computers*, Volume C-29, pp. 360-371, May 1980.

[Gries77]
David Gries, "An Exercise in Proving Programs Correct", *Communications of the ACM*, Volume 20, pp.921-930, December 1977.

[Harris77]
J. A. Harris and D. R. Smith, "Hierarchical Multiprocessor Organizations", *Proceeding of the 4th Symposium on Computer Architecture*, pp.41-48, March 1977.

[Hewitt77]
Carl Hewitt, "Viewing Control Structures as Patterns for Passing Messages", *Artificial Intelligence*, Volume 8, pp.323-364, 1977.

[Hewitt80]
Carl Hewitt, "The Apiary Network Architecture for Knowledgeable Systems", *Conference Record of the 1980 LISP Conference*, Stanford University, August 1980.

[Hoare'78]
C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, Volume 21, Number 8, pp. 666-677, August 1978.

[Horowitz81]
E. Horowitz and A. Zorat, "The Binary Tree as an Interconnection Network: Applications to Multiprocessor System and VLSI", *IEEE Transactions on Computers*, Volume C-30, pp. 247-264, April 1981.

[Ingalls78]
Dan Ingalls, "The Smalltalk 76 Programming System: Design and Implementation", *Proceedings of the Fifth ACM Conference on Principles of Programming Systems*, pp. 9-16, January 1978.

[Johnsson81]
Lennart Johnsson, "Computational Arrays for Band Matrix Problems", Department of Computer Science, Display File 4287, California Institute of Technology, May 1981.

[Jones73]
Anita K. Jones, "Protection in Programmed Systems", Ph.D Thesis, Department of Computer Science, Carnegie-Mellon University, 1973.

[Kahn81]
Kevin C. Kahn, William M Corwin, T. Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, Fred J. Pollack and Michael R. Gifkins, "iMAX: A Multiprocessor Operating System for an Object-Based Computer", *Operating Systems Review*, ACM SIGOPS, December 1981.

[Kung78]
H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)", Technical Report, Department of Computer Science, Carnegie-Mellon University, December 1978.

[Lang76]
T. Lang, "Interconnections Between Processors and Memory Using the Shuffle/Exchange Network", *IEEE Transactions on Computers*, Volume C-25, pp.496-503, May 1976.

[Lawrie73]
D. H. Lawrie, "Memory-Processor Connection Networks", Department of Computer Science, University of Illinois, Report 557, February 1973.

[Liskov77]
Barbara Liskov, A. Snyder, R. Atkinson and C. Schaffert, "Abstraction Mechanisms in CLU", *Communications of the ACM*, Volume 20, Number 8, pp. 564-576, August 1977.

[Locanthi80]
Bart Locanthi, "The Homogeneous Machine", Ph.D Dissertation, Computer Science Department, California Institute of Technology, 1980.

[Mago79]
Gyula Mago, "A Cellular, Language Directed Architecture", *Proceedings of the Caltech Conference on Very Large Scale Integration*, pp.435-445, January 1979.

[Martin81]
Alain Martin, "The Torus: An Exercise in Constructing a Processing Surface", *Proceedings of the Second Caltech Conference on Very Large Scale Integration*, January 1981.

[McCarthy60]
J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine", *Communications of the ACM*, Volume 3, pp.184-195, April 1960.

[Metcalfe76]
R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, Volume 19, pp.395-403, July 1976.

[Moore79]
Gordon E. Moore, "Are We Really Ready for VLSI?", *Proceedings of the Caltech Conference on Very Large Scale Integration*, pp.3-14, January 1979.

[Newman79]
William M. Newman and Robert F. Sproull, "Chapter5: Clipping and Windowing", *Principles of Interactive Computer Graphics*, McGraw-Hill, Inc., 1979.

[Ousterhout80]
J. K. Ousterhaut et al, "Medusa: An Experiment in Distributed Operating System Structure", *Communications of the ACM*, Volume 23, pp. 92-105, February 1980.

[Owicki80]
Susan Owicki and Leslie Lamport, "Proving Liveness Properties of Concurrent Programs", Computer Systems Laboratory, Stanford University, October 1980.

[Parker80]
D. S. Parker, "Notes on Shuffle/Exchange-Type Switching Networks", *IEEE Transactions on Computers*, Volume C-29, pp.213-222, March 1980.

[Pease75]
M. C. Pease, "The Indirect Binary N-Cube Microprocessor Array", *IEEE Transactions on Computers*, Volume C-26, pp.458-473, May 1977.

[Pippenger75]
N. Pippenger, "On Crossbar Switching Networks", *IEEE Transactions on Communication*, Volume COM-23, pp.646-659, June 1975.

[Rice71]
R. Rice and W. R. Smith, "SYMBOL - A Major Departure from Classic Software Dominated von Neumann Computing Systems", *Proceeding of the AFIPS Spring Joint Computer Conference*, Volume 36, pp.575-587, Spring 1971.

[Ritchie74]
D. M. Ritchie and K. L. Thompson, "The UNIX Timesharing System", *Communications of the ACM*, July 1974.

[Seitz82]
Charles L. Seitz, "Ensemble Architectures for VLSI - A Survey and Taxonomy", *Proceedings of the Conference on Advanced Research in VLSI*, Massachusetts Institute of Technology, pp.130-135, January 1982.

[Snyder79]
Alan Snyder, "A Machine Architecture to Support Object Oriented Languages", Ph.D Thesis, Massachusetts Institute of Technology,

Laboratory for Computer Science, TR-209, March 1979.

[Spier69]
M. Spier and E. Organick, "The Multics Interprocess Communication Facility", *Proceedings of the 2nd Symposium on Operating Systems Principles*, Princeton University, October 1969.

[Steele75]
G. L. Steele, "Multiprocessing Compactifying Garbage Collection", *Communications of the ACM*, Volume 18, pp.495-508, September 1975.

[Stone71]
H. S. Stone, "Parallel Processing with the Perfect Shuffle", *IEEE Transactions on Computers*, Volume C-20, pp.153-161, February 1971.

[Sullivan77]
H. Sullivan and T. R. Bashkow, "A Large Scale Homogeneous, Fully Distributed Parallel Machine I", *Proceedings of the 4th Symposium on Computer Architecture*, pp. 105-117, March 1977.

[Sutherland77]
I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science", *Scientific American*, pp.210-228, September 1977.

[Thompson78]
C. D. Thompson, "Generalized Connection Networks for Parallel Processor Intercommunication", *IEEE Transactions on Computers*, Volume C-27, pp. 1119-1125, December 1978.

[Thornton70]
J. G. Thornton, *Design of a Computer: The Control Data 6600*, Scott, Foresman and Co., 1970.

[Thurber74]
K. J. Thurber, "Interconnection Networks - A Survey and Assessment", *AFIPS Conference Proceedings*, Volume 43, pp. 909-919, NCC 1974.

[Treleaven80]
P. C. Treleaven and R. P. Hopkins, "Decentralised Computation", Computing Laboratory, University of Newcastle upon Tyne, Number 157, November 1980.

[Van Wijngaarden69]
A. Van Wijngaarden, ed. B. J. Maillous, J. Peck and C. Koster, *Report of the Algorithmic Language ALGOL 68*, Springer-Verlag, 1969.

[Wadler76]
Philip L. Wadler, "Analysis of an Algorithm for Real Time Garbage Collection", *Communications of the ACM*, Volume 19, pp.491-500, September 1976.

[Wittie76]
L. D. Wittie, "Efficient Message Routing in Mega-micro-computer Networks", *Proceedings of the 3rd Symposium on Computer Architecture*, pp. 136-140, January 1976.

[Wittie81]
L. D. Wittie, "Communication Structures for Large Networks of Microcomputers", *IEEE Transactions on Computers*, Volume C-30, pp. 264-273, April 1981.

[Wu80]
S. B. Wu, *Interconnection Design and Resource Assignment for Large Multi-microcomputer Networks*, Ph.D dissertation, Department of Computer Science, Ohio State University, Columbus, Ohio, December 1980.

[Wulf72]
W. A. Wulf and C. G. Bell, "C.mmp - A Multi-Mini-Processor", *Proceedings of AFIPS Fall Joint Computer Conference*, Volume 41, Part II, pp.765-777, Fall 1972.

[Wulf76]
W. Wulf, R. London and M. Shaw, "Abstraction and Verification in ALPHARD", Department of Computer Science, Carnegie-Mellon University, 1976.

[Wulf80]
William A. Wulf, Samuel Harbison and Roy Levin, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, 1980.

[Yu81]
Kwang-I Yu, "Communicative Databases", Ph.D Thesis, Computer Science Department,California Institute of Technology, 1981.

Appendix A

Network Simulation Results

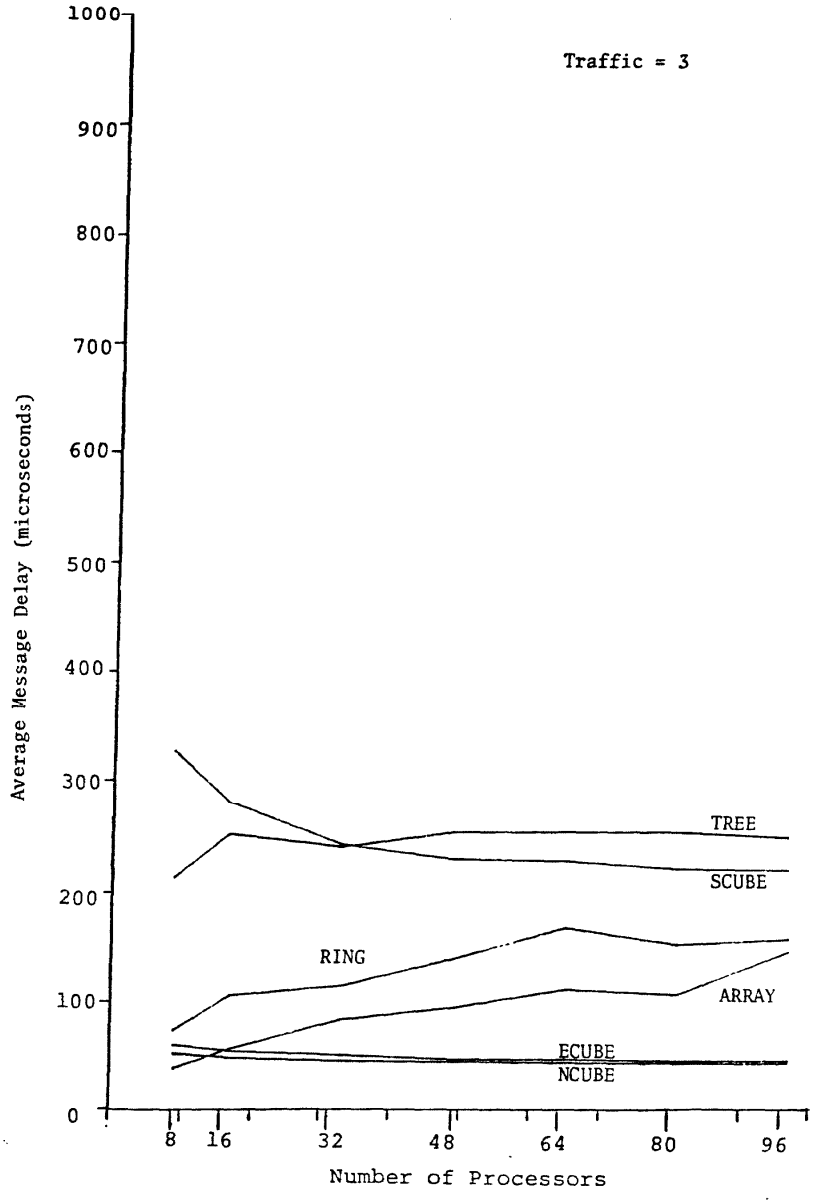


Figure A-1

Average Message Delay vs. Network Size (a=3)

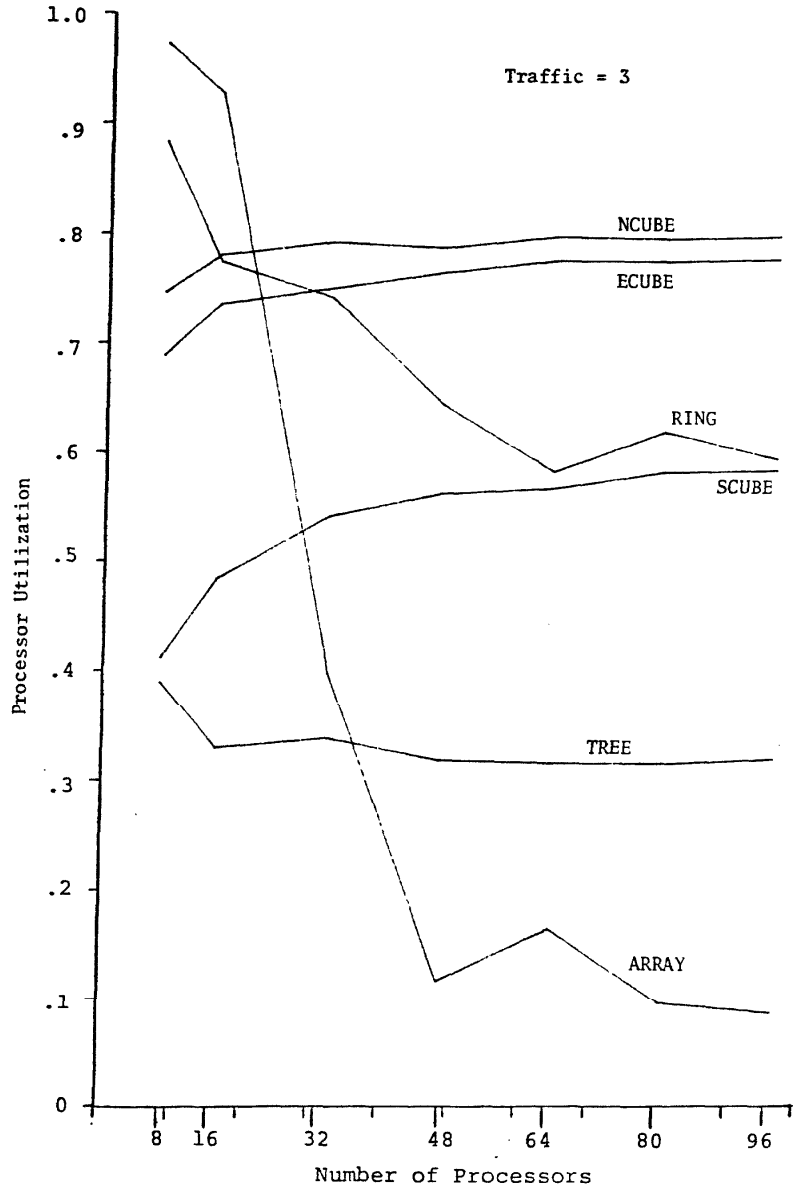


Figure A-2

Processor Utilization vs. Network Size (a=3)

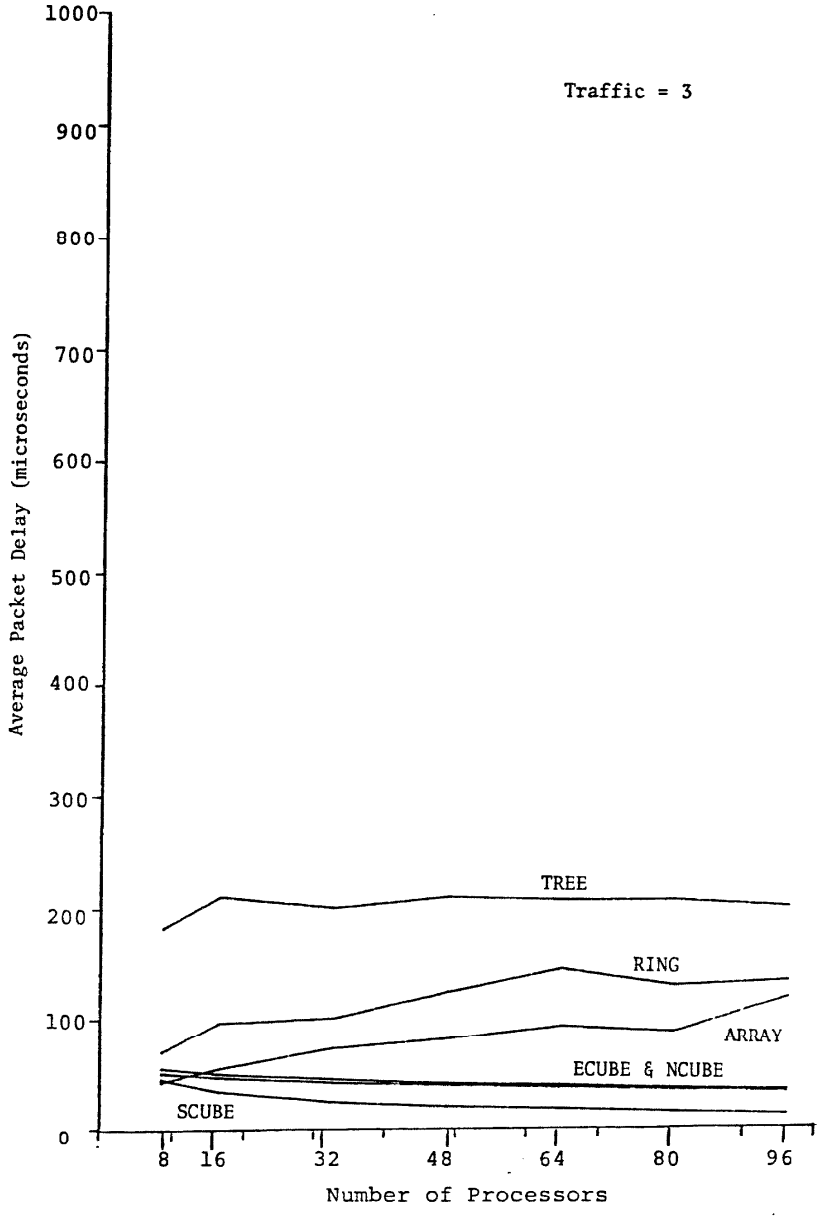


Figure A-3

Average Packet Delay vs. Network Size (a=3)

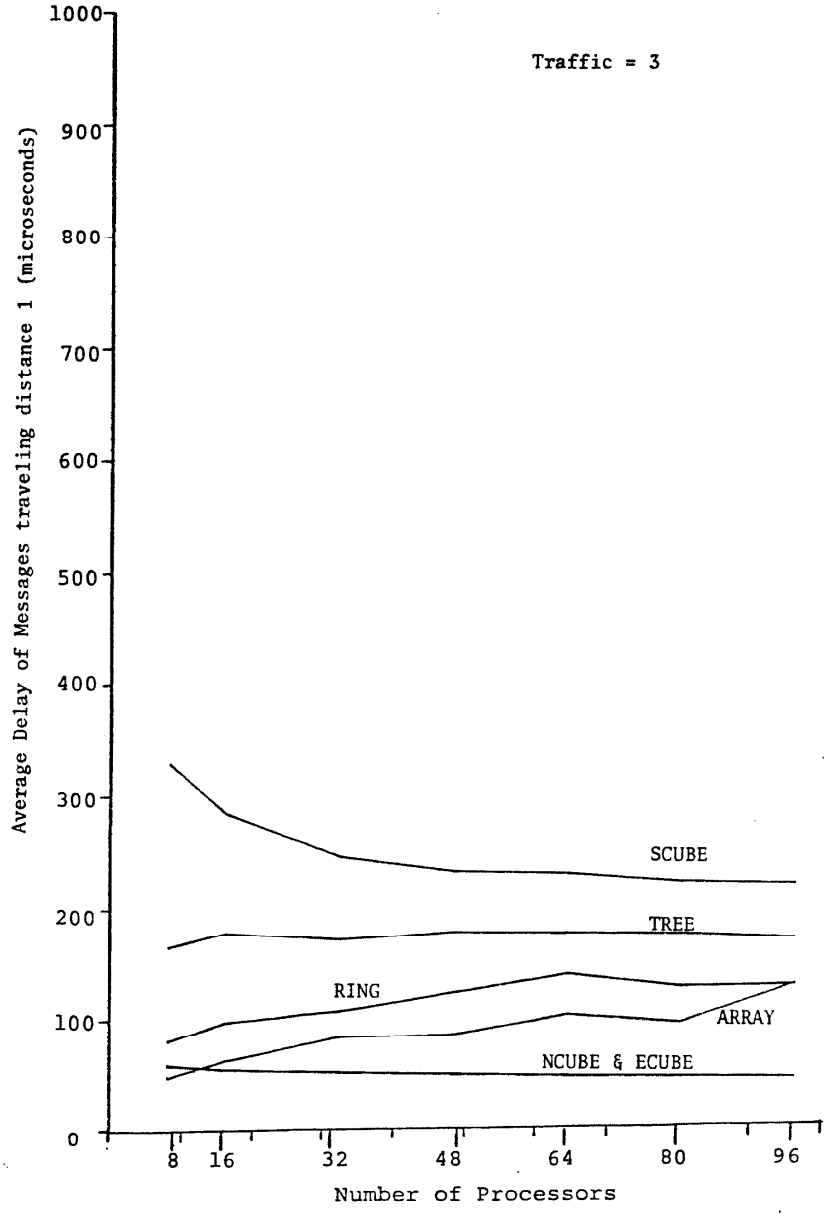


Figure A-4

Local Message Delay vs. Network Size (a=3)

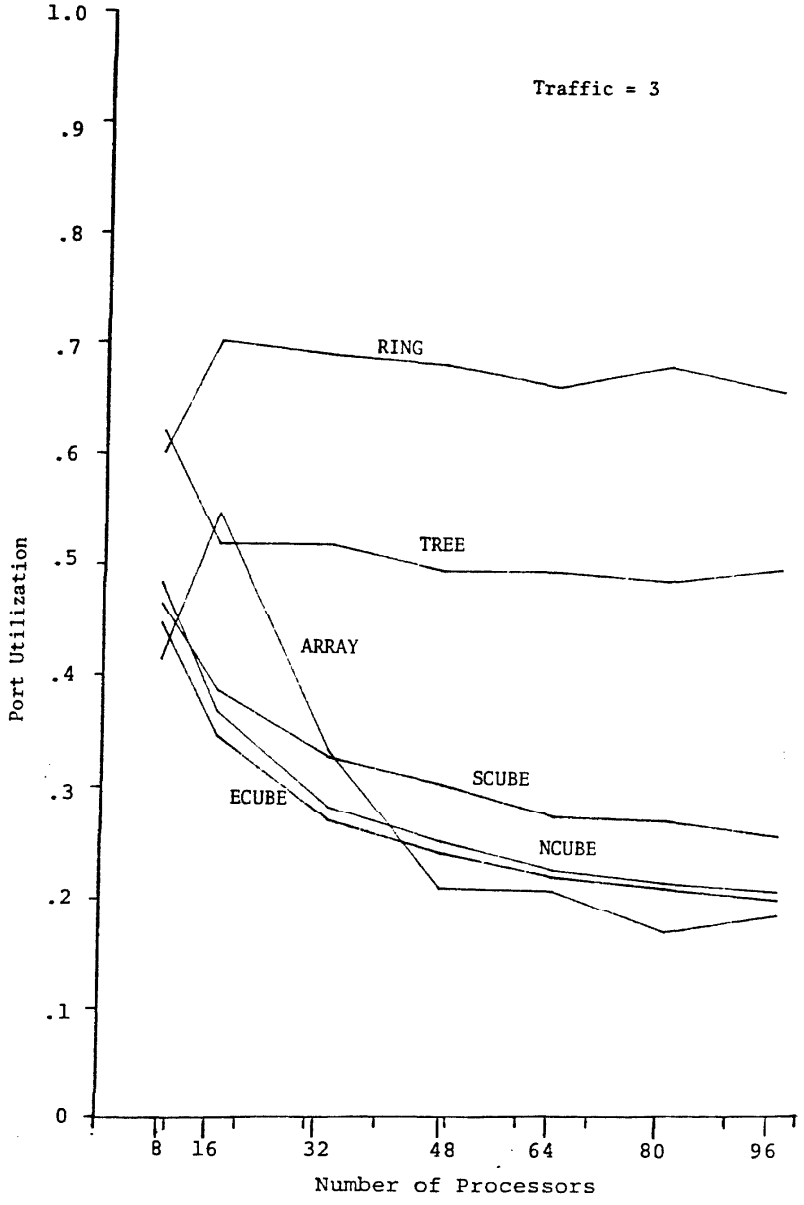


Figure A-5

Comm. Link Utilization vs. Network Size (a=3)

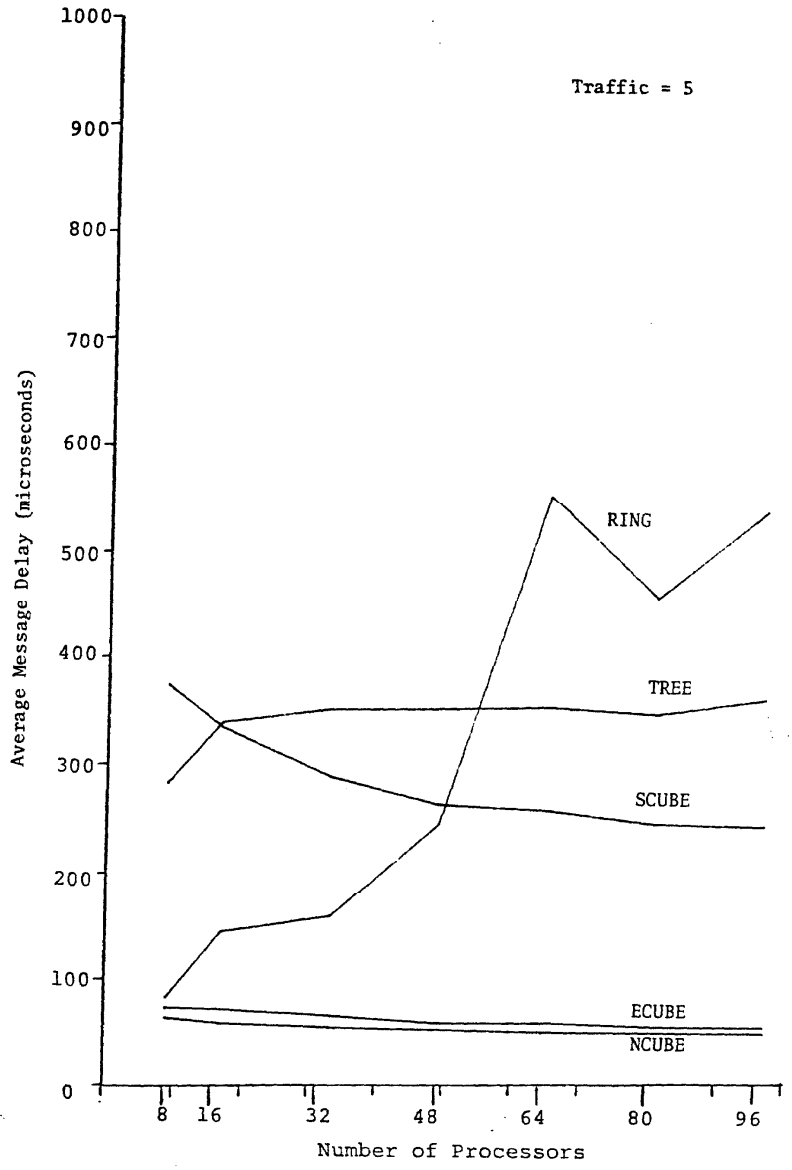


Figure A-6

Average Message Delay vs. Network Size (a=5)

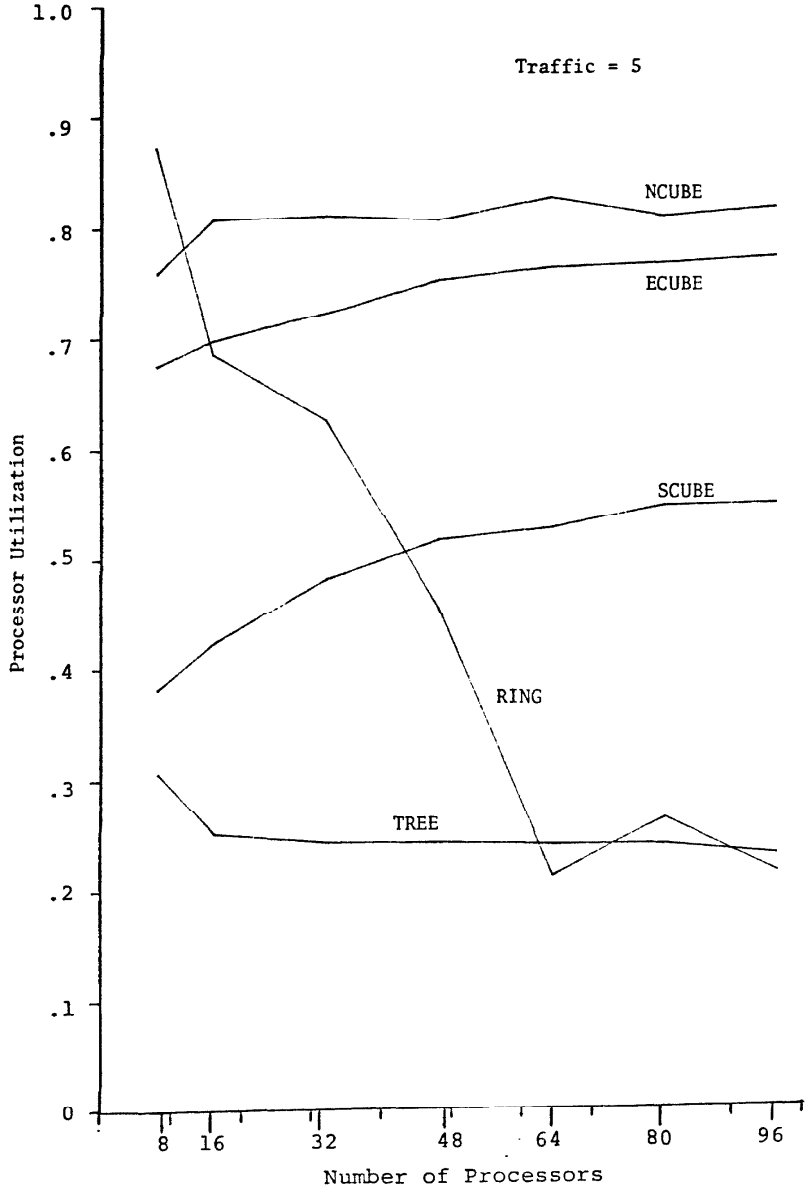


Figure A-7

Processor Utilization vs. Network Size (a=5)

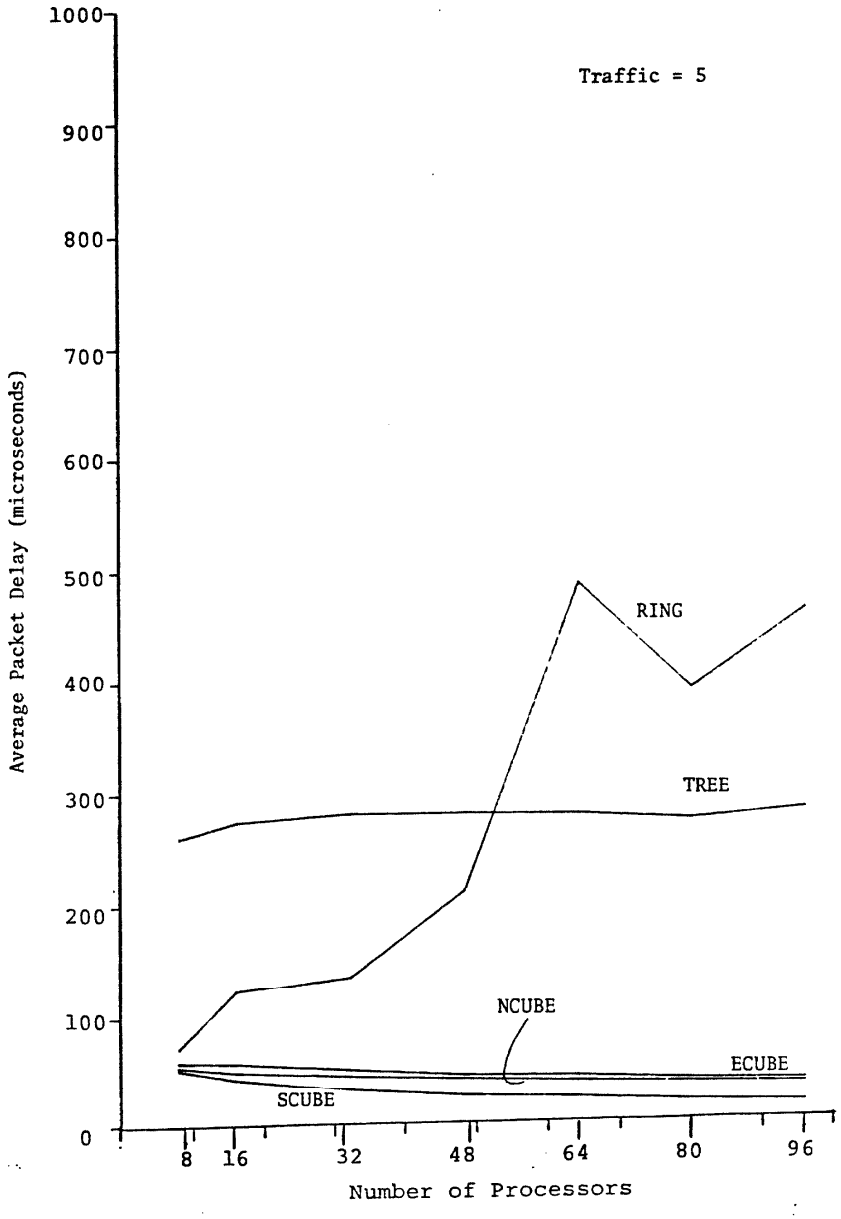


Figure A-8

Average Packet Delay vs. Network Size (a=5)

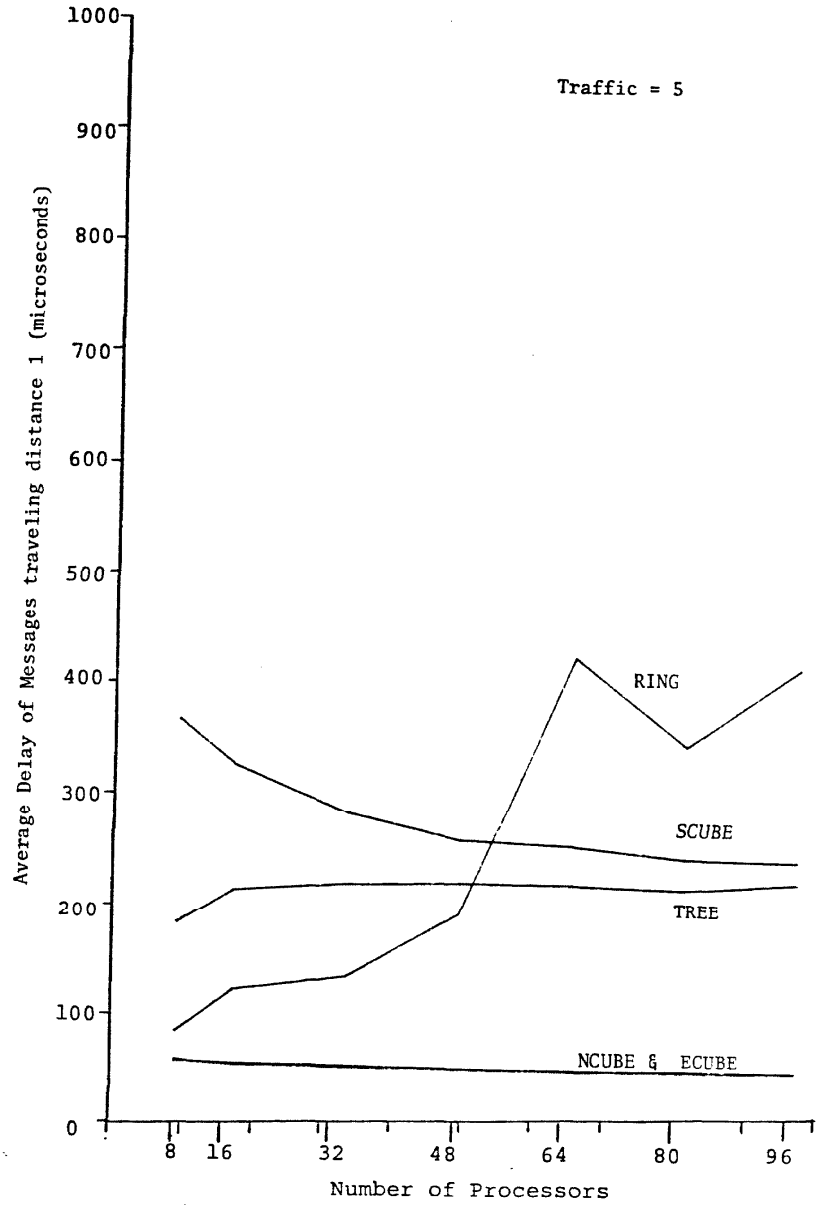


Figure A-9

Local Message Delay vs. Network Size (a=5)

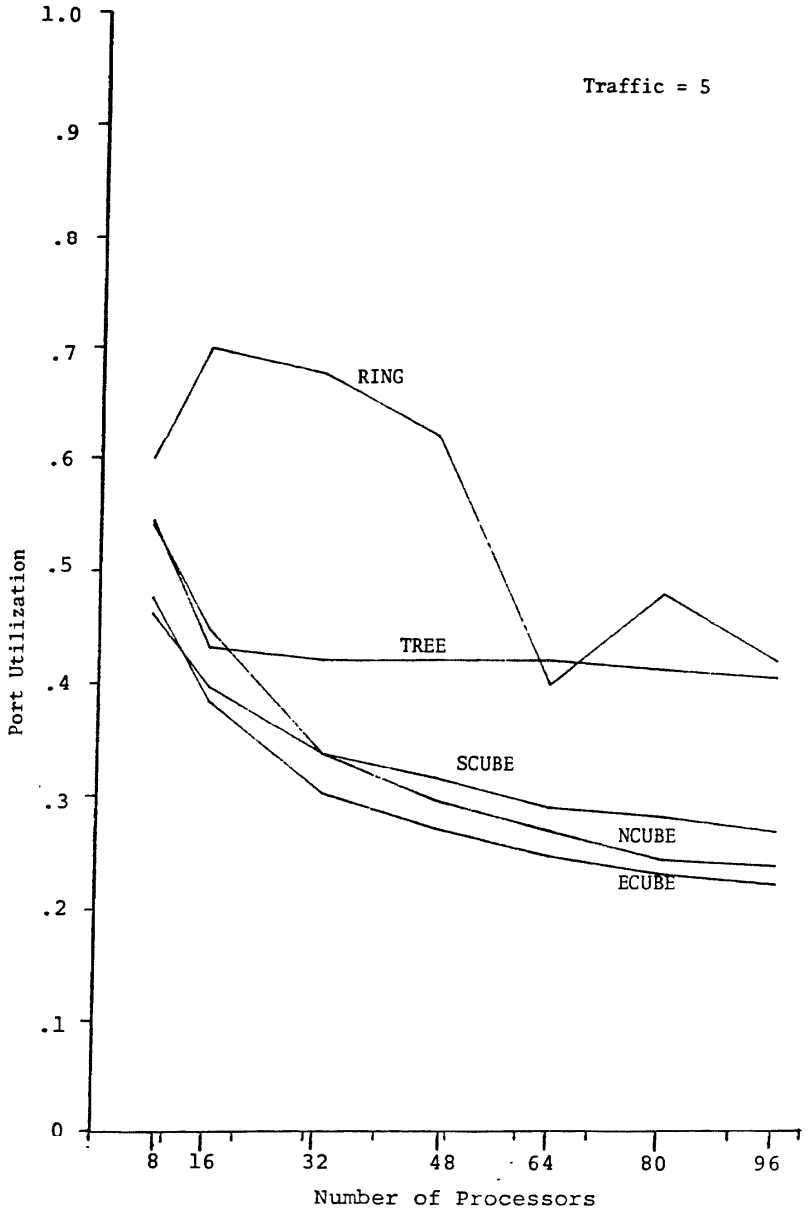


Figure A-10

Comm. Link Utilization vs. Network Size (a=5)

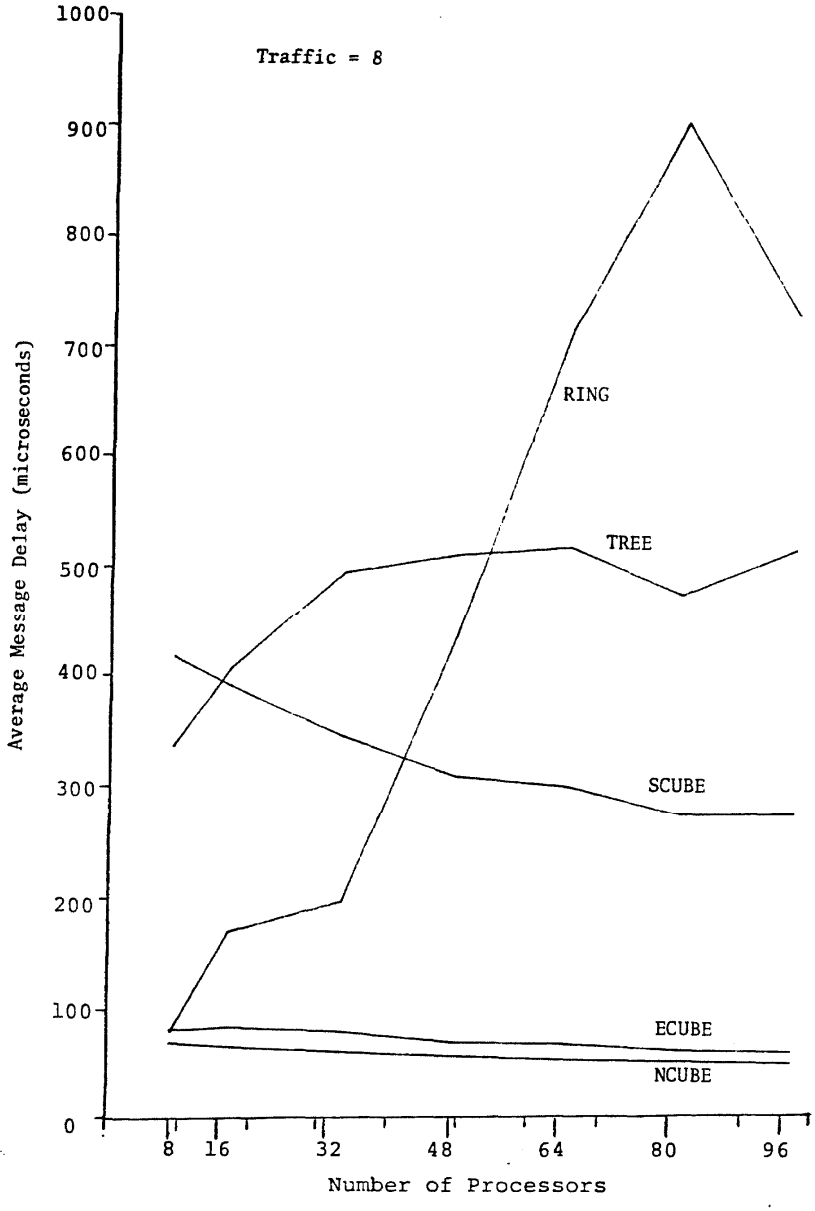


Figure A-11

Average Message Delay vs. Network Size (a=8)

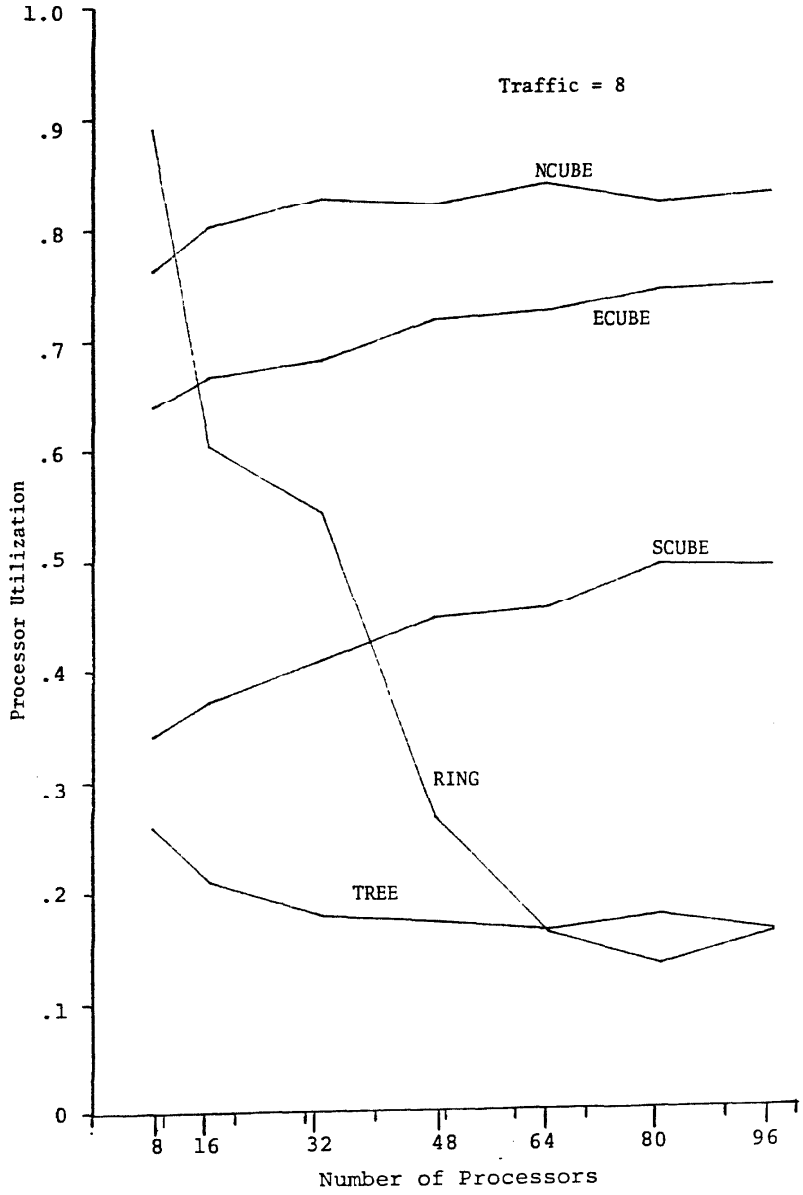


Figure A-12

Processor Utilization vs. Network Size (a=8)

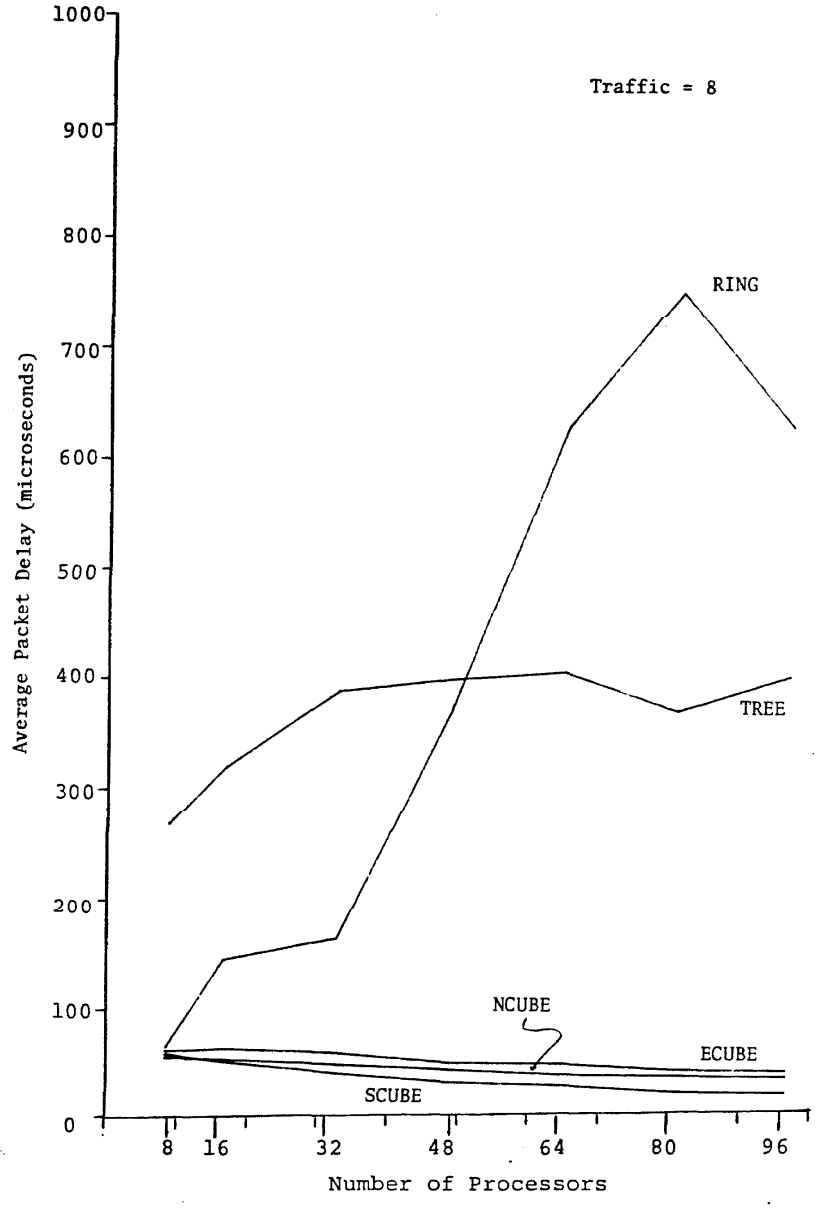


Figure A-13

Average Packet Delay vs. Network Size (a=8)

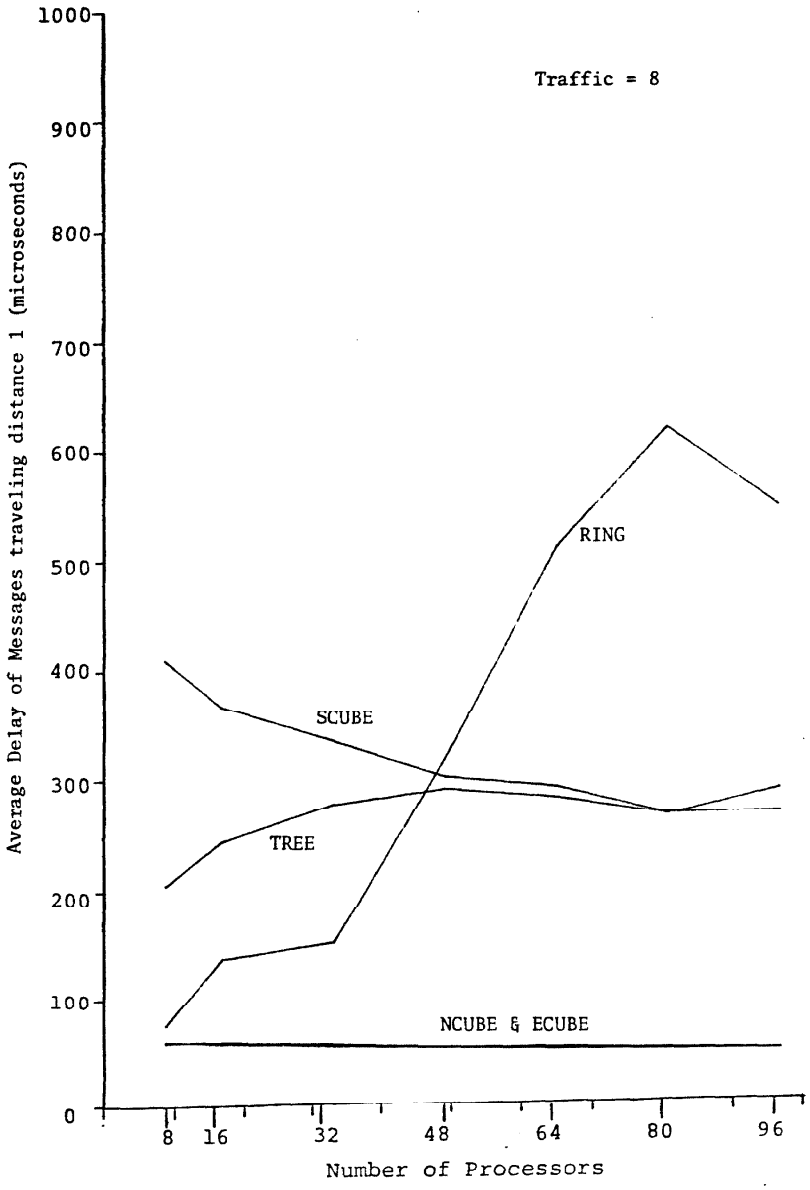


Figure A-14

Local Message Delay vs. Network Size (a=8)

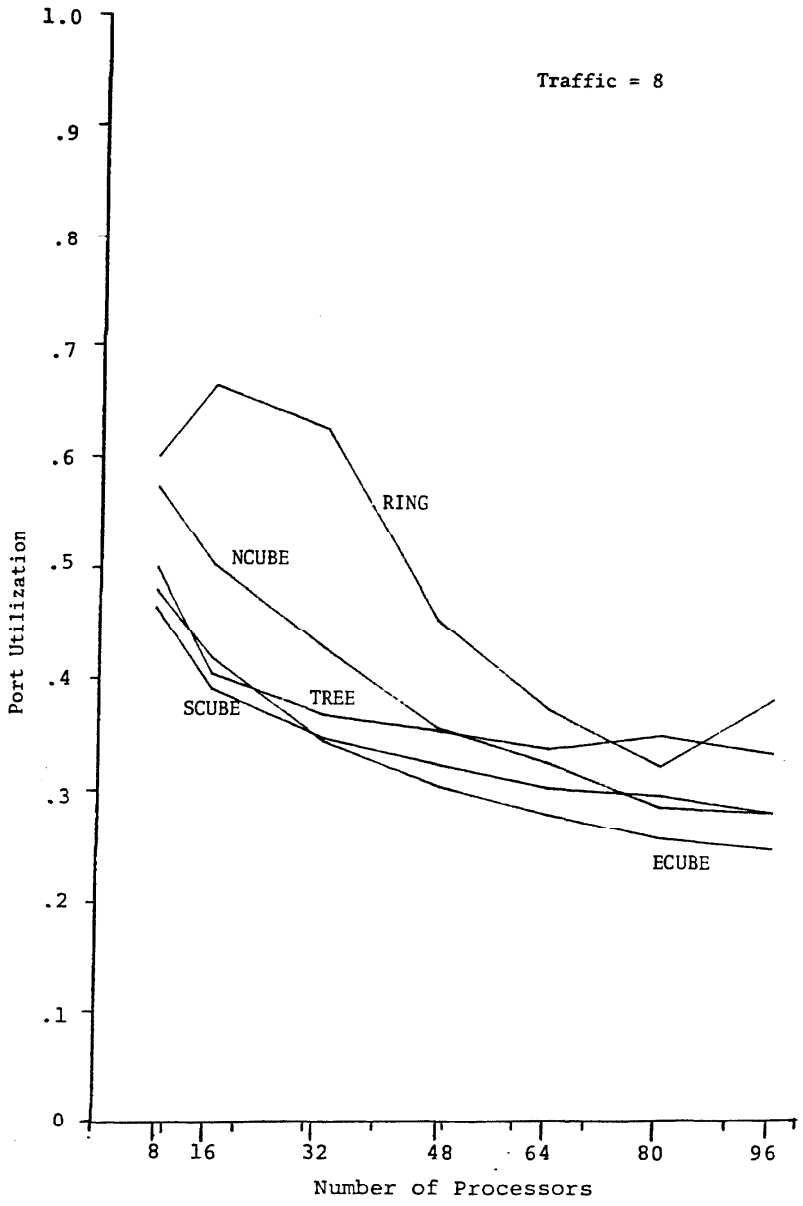


Figure A-15

Comm. Link Utilization vs. Network Size (a=8)

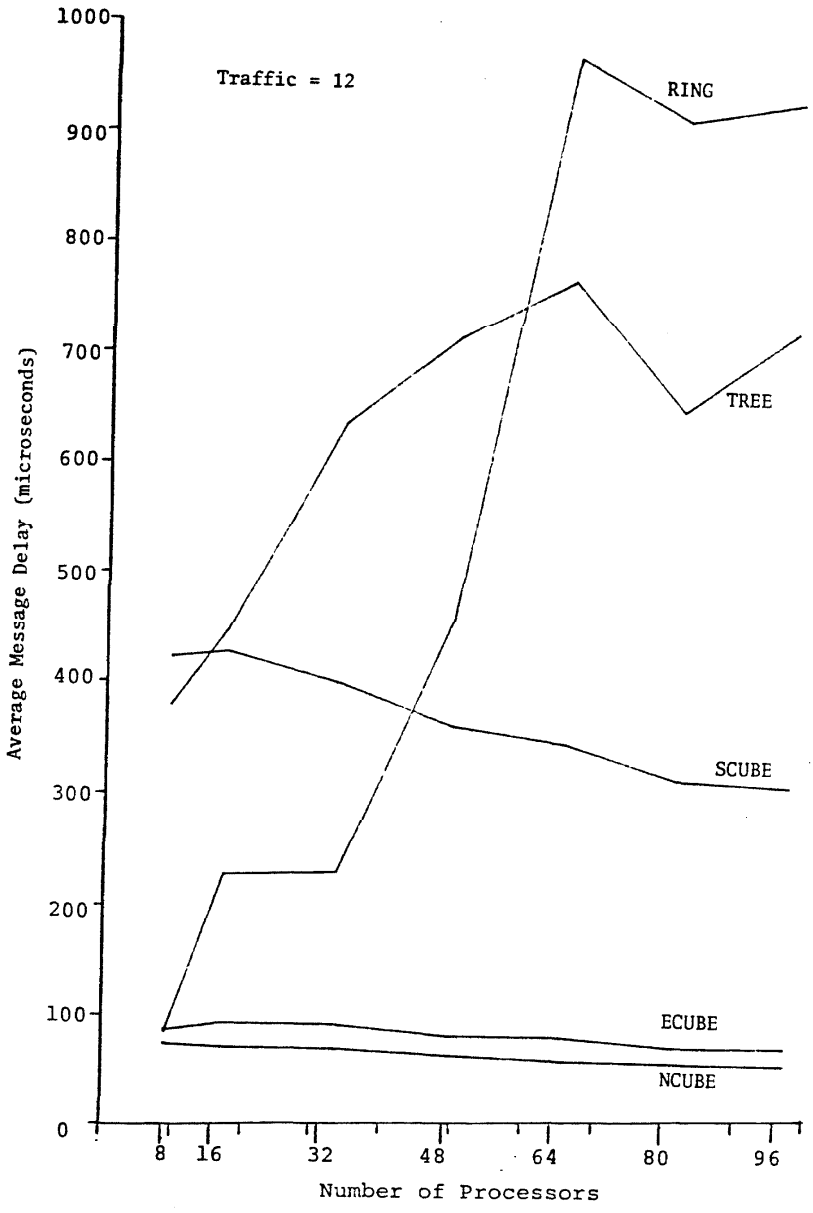


Figure A-16

Average Message Delay vs. Network Size (a=12)

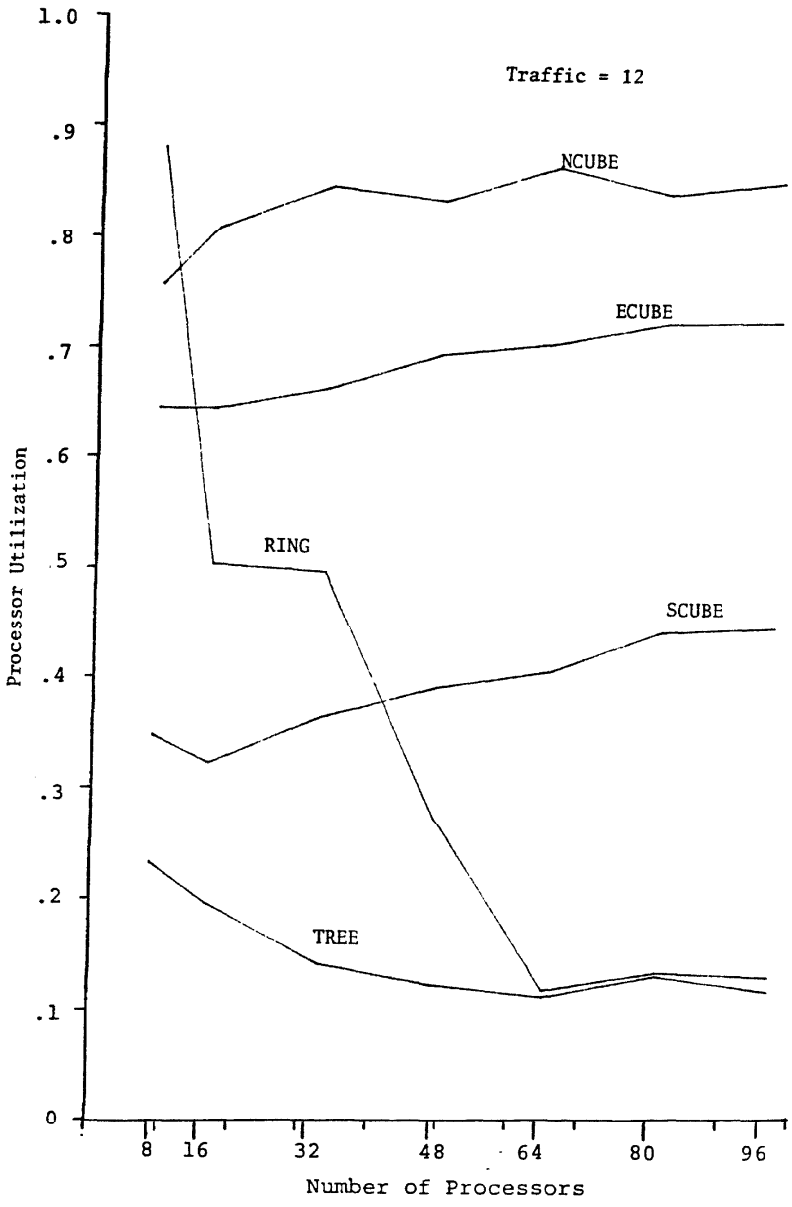


Figure A-17

Processor Utilization vs. Network Size (a=12)

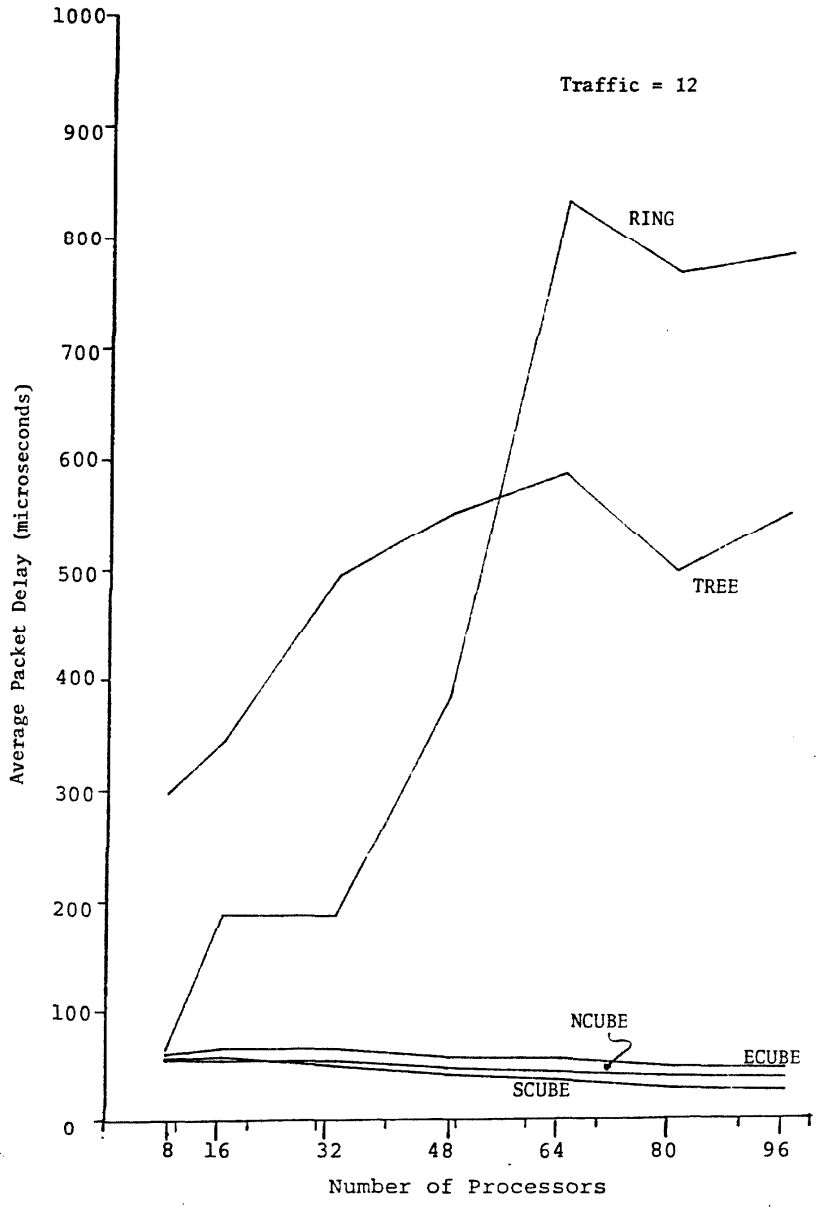


Figure A-18

Average Packet Delay vs. Network Size (a=12)

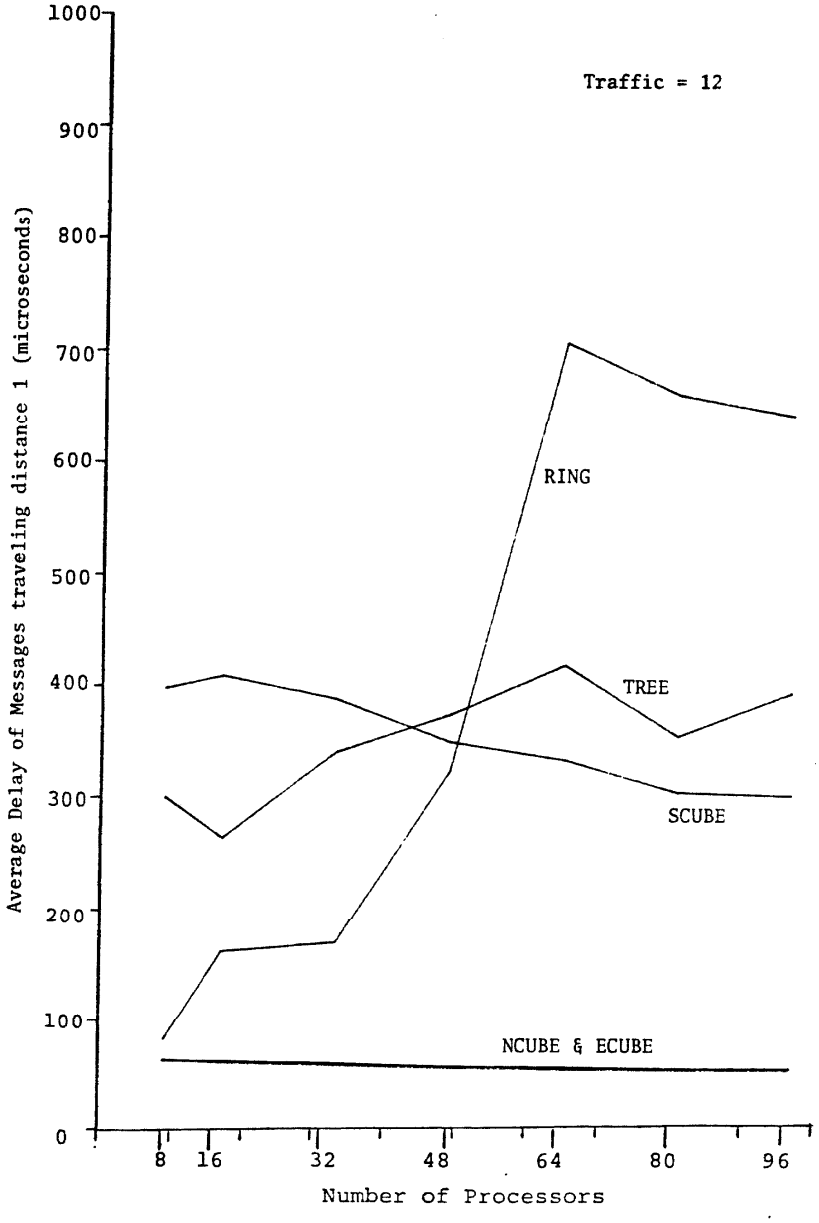


Figure A-19

Local Message Delay vs. Network Size (a=12)

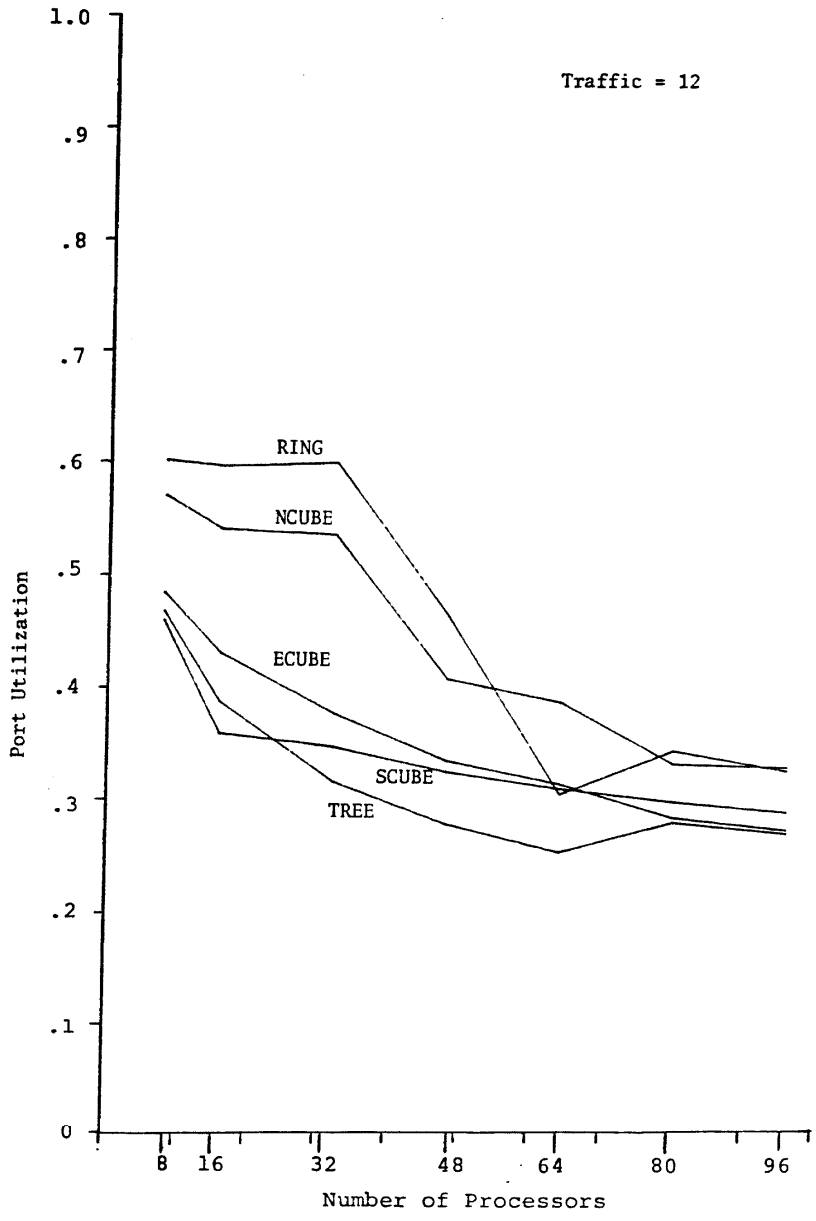


Figure A-20

Comm. Link Utilization vs. Network Size (a=12)

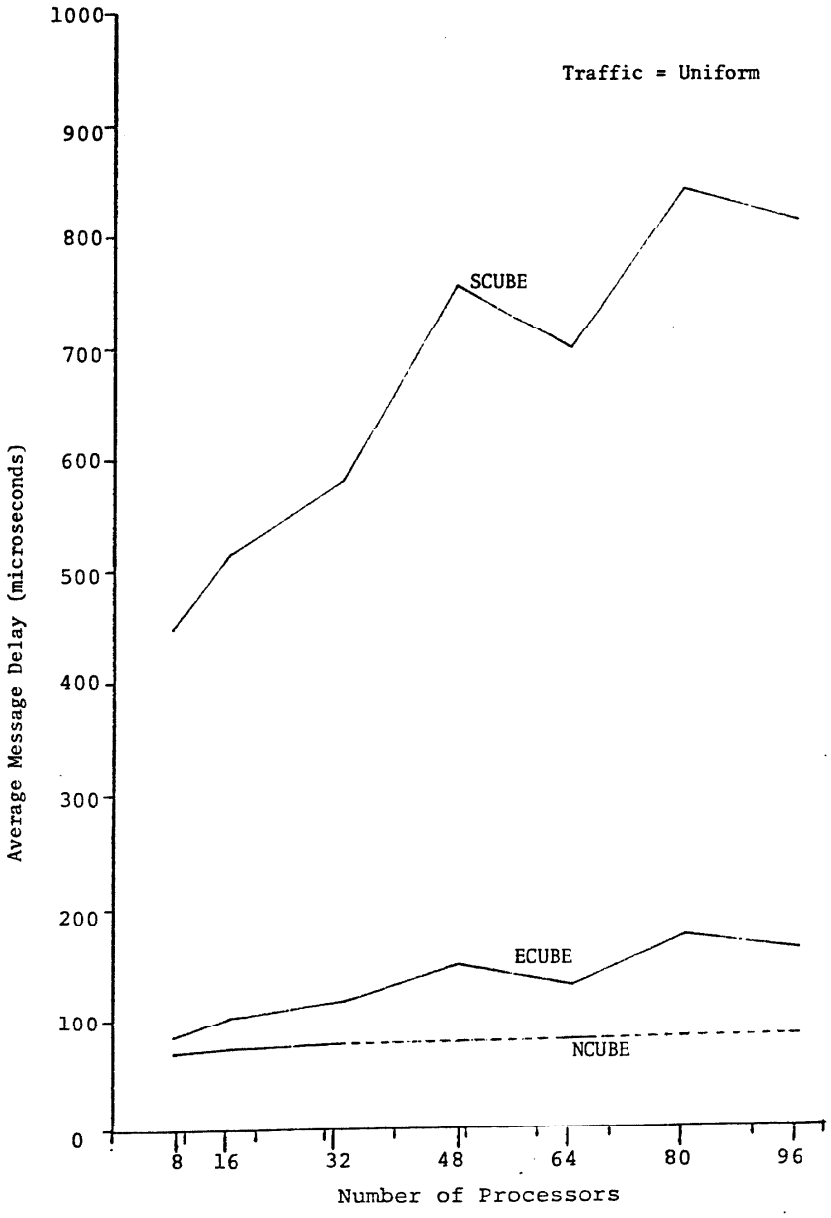


Figure A-21

Avg. Message Delay vs. Network Size (Unif. Traf)

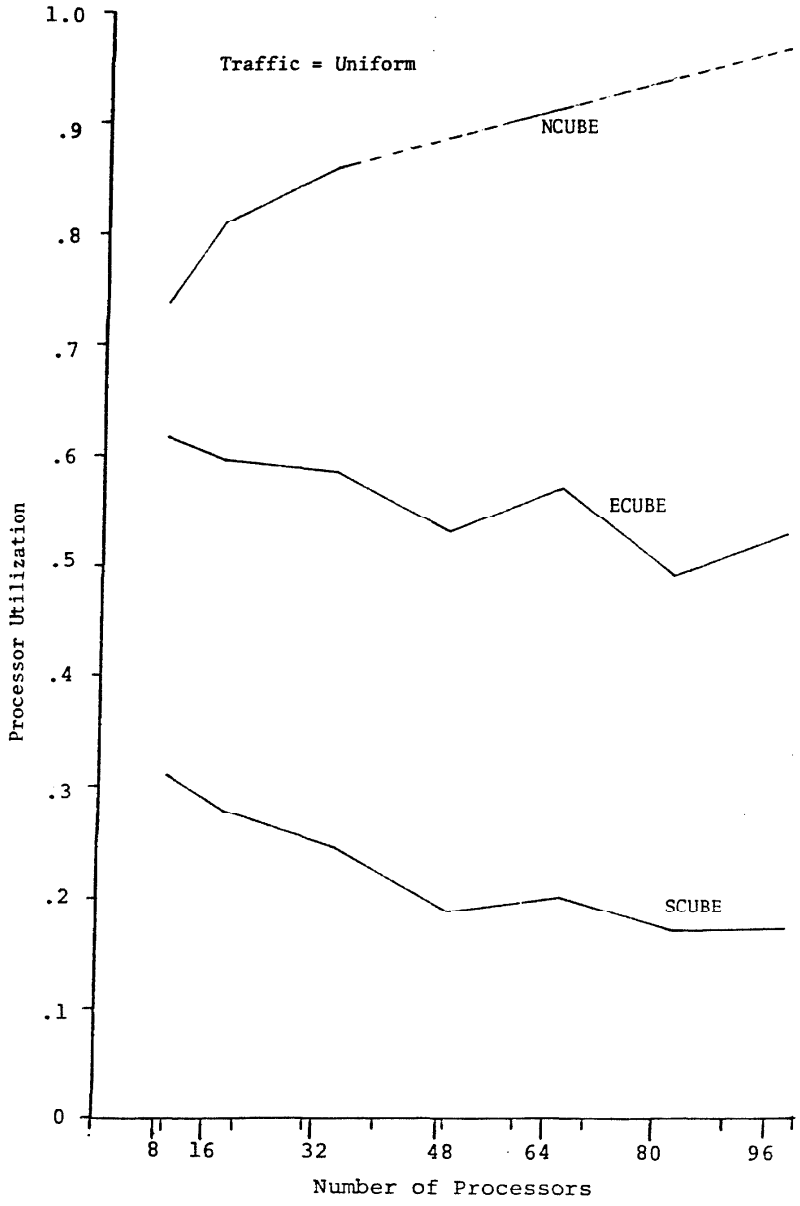


Figure A-22

Proc. Utilization vs. Network Size (Unif. Traf)

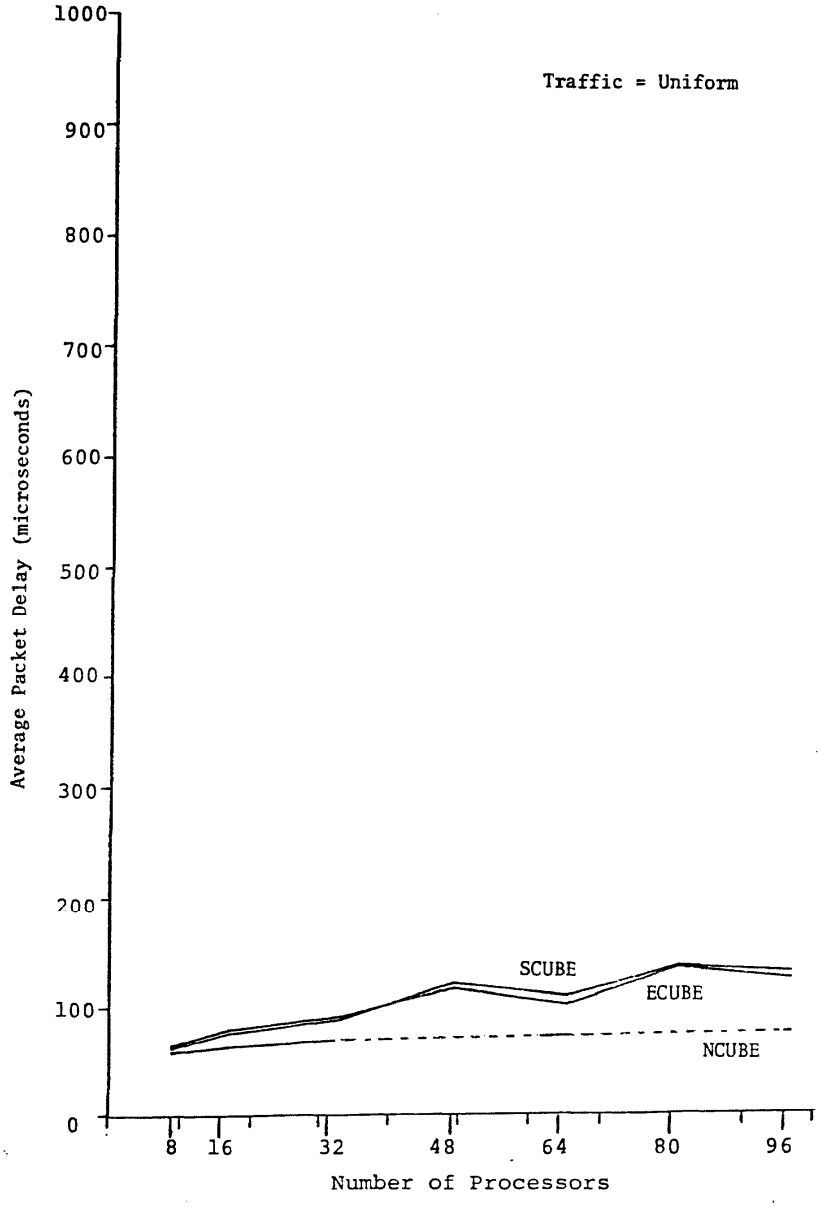


Figure A-23

Avg. Packet Delay vs. Network Size (Unif. Traf)

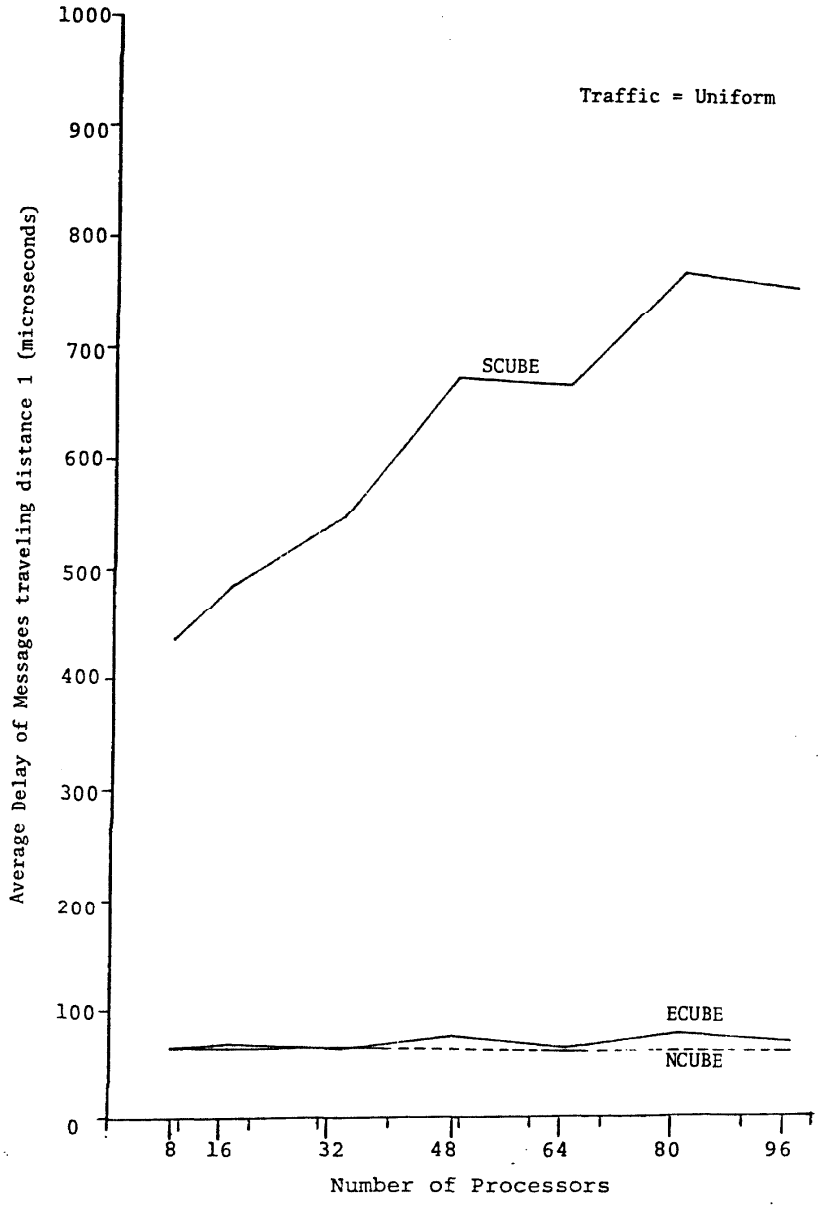


Figure A-24

Local Msg. Delay vs. Network Size (Unif. Traf)

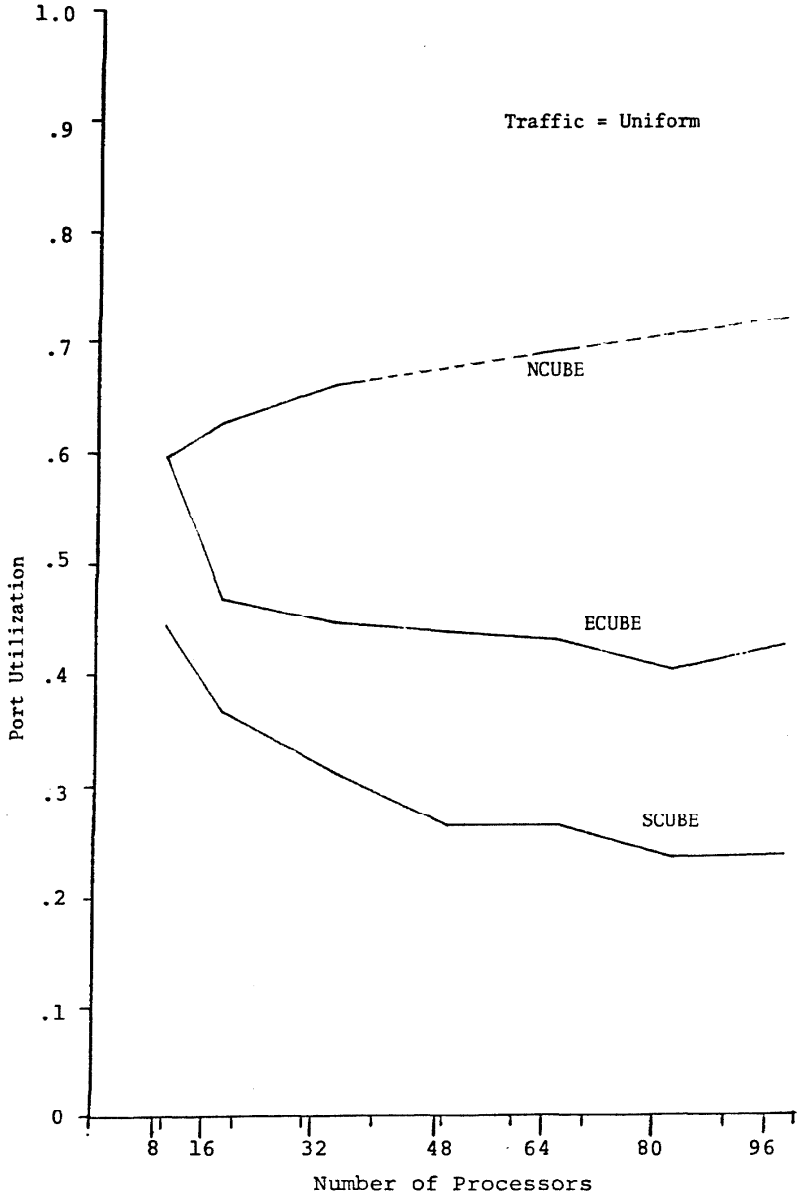


Figure A-25

Comm. Link Util. vs. Network Size (Unif. Traf)

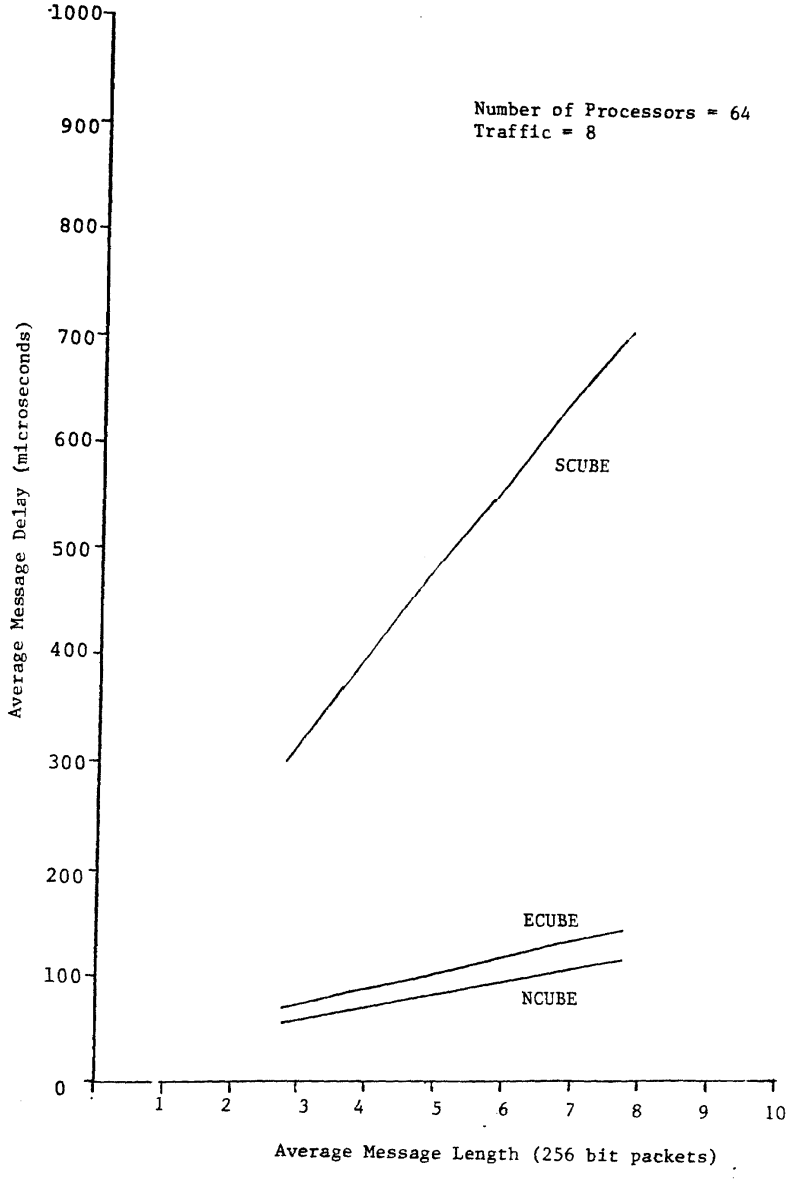


Figure A-26

Avg. Msg. Delay vs. Message Length (64 Proc.,a=8)

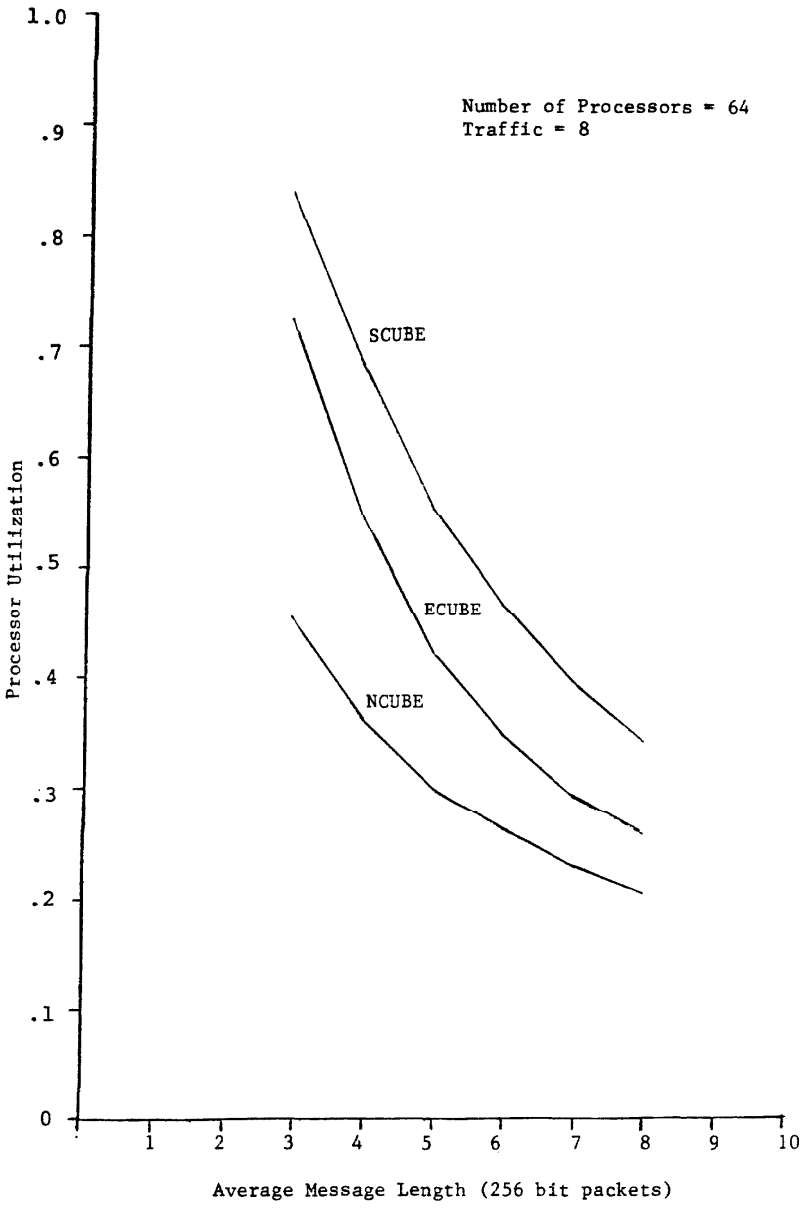


Figure A-27

Proc. Util. vs. Message Length (64 Proc.,a=8)

Appendix B

Example Network Simulator Output

Boolean N-cube Connection Run on: 1981-06-05 at 08:00:52
Number of Processors: 64 (Dimension=6)
Communication Link Data Rate: 20 Megabits/sec
Packet Size: 256
Number of Packets of Storage in Port Queue: 4

CLOCK TIME = 1.000E-02

****
* REPORT *
****

DISTRIBUTIONS

TITLE /	(RE)SET/	OBS/	TYPE /	A/	B/	SEED
MsgLength	2.000E-03	14373	NORMAL	768.000	256.000	33427485
MsgFrequency	2.000E-03	14378	NORMAL	3.000E-05	1.500E-05	22276755
Proc Source	2.000E-03	14380	UNIFORM	0.000	0.125	46847980

ACCUMULATES

TITLE /	(RE)SET/	OBS/	AVERAGE/	EST.ST.DV/	MINIMUM/	MAXIMUM
Proc Active %	2.000E-03	28751	84.428	4.548	65.625	98.438
Port Active %	2.000E-03	156937	32.696	2.623	23.698	40.885
Port Q Length	2.000E-03	156931	0.604	4.728E-02	0.443	0.784
Transit Packs	2.000E-03	100635	244.971	19.032	177.000	314.000
Transit Msgs	2.000E-03	28753	92.215	6.596	70.000	118.000

HISTOGRAMS

SUMMARY

TITLE /	(RE)SET/	OBS/	AVERAGE/	EST.ST.DV/	MINIMUM/	MAXIMUM
Proc Select	2.000E-03	14373	8.600	8.074	0.500	61.967

CELL/	LOWER LIM/	N/	FREQ/	CUM :	
0	-INFINITY	0	0.00	0.00	I
1	0.500	1025	0.07	7.13	I***************
2	1.120	1008	0.07	14.14	I***************
3	1.740	912	0.06	20.49	I*************
4	2.360	854	0.06	26.43	I************
5	2.980	814	0.06	32.09	I************
6	3.600	690	0.05	36.90	I**********
7	4.220	662	0.05	41.50	I*********

8	4.840	628	0.04	45.87	I*********
9	5.460	578	0.04	49.89	I********
10	6.080	513	0.04	53.46	I*******
11	6.700	502	0.03	56.95	I*******
12	7.320	461	0.03	60.16	I*******
13	7.940	450	0.03	63.29	I******
14	8.560	370	0.03	65.87	I*****
15	9.180	343	0.02	68.25	I*****
16	9.800	351	0.02	70.70	I*****
17	10.420	286	0.02	72.68	I****
18	11.040	274	0.02	74.59	I****
19	11.660	289	0.02	76.60	I****
20	12.280	256	0.02	78.38	I****
21	12.900	232	0.02	80.00	I***
22	13.520	231	0.02	81.60	I***
23	14.140	191	0.01	82.93	I***
24	14.760	175	0.01	84.15	I**
25	15.380	167	0.01	85.31	I**
26	16.000	2111	0.15	100.00	I******************************

S U M M A R Y

TITLE /	(RE)SET/	OBS/	AVERAGE/EST.ST.DV/		MINIMUM/	MAXIMUM
Msg Distance	2.000E-03	14380	1.560	0.654	1.000	5.000

CELL/LOWER LIM/	N/	FREQ/	CUM :	
0 -INFINITY	0	0.00	0.00	I
1 0.000	0	0.00	0.00	I
2 1.000	7509	0.52	52.22	I******************************
3 2.000	5793	0.40	92.50	I**********************
4 3.000	979	0.07	99.31	I****
5 4.000	90	0.01	99.94	I.
6 5.000	9	0.00	100.00	I.
7 6.000	0	0.00	100.00	I

S U M M A R Y

TITLE /	(RE)SET/	OBS/	AVERAGE/EST.ST.DV/		MINIMUM/	MAXIMUM
Msg Length	2.000E-03	14373	766.830	255.279	-194.000	1698.000

CELL/LOWER LIM/	N/	FREQ/	CUM :	
0 -INFINITY	17	0.00	0.12	I.
1 0.000	22	0.00	0.27	I.
2 61.440	44	0.00	0.58	I*
3 122.880	86	0.01	1.18	I**
4 184.320	128	0.01	2.07	I***
5 245.760	234	0.02	3.69	I*****
6 307.200	338	0.02	6.05	I*******
7 368.640	513	0.04	9.62	I**********
8 430.080	619	0.04	13.92	I************
9 491.520	857	0.06	19.88	I******************

10	552.960	1077	0.07	27.38	I**********
11	614.400	1243	0.09	36.03	I***********
12	675.840	1339	0.09	45.34	I************
13	737.280	1325	0.09	54.56	I************
14	798.720	1387	0.10	64.21	I*************
15	860.160	1251	0.09	72.91	I***********
16	921.600	1069	0.07	80.35	I**********
17	983.040	829	0.06	86.12	I********
18	1044.480	646	0.04	90.61	I******
19	1105.920	516	0.04	94.20	I*****
20	1167.360	347	0.02	96.62	I***
21	1228.800	209	0.01	98.07	I**
22	1290.240	119	0.01	98.90	I*
23	1351.680	82	0.01	99.47	I*
24	1413.120	36	0.00	99.72	I
25	1474.560	26	0.00	99.90	I
26	1536.000	14	0.00	100.00	I.

S U M M A R Y

TITLE /	(RE)SET/	OBS/	AVERAGE/	EST.ST.DV/	MINIMUM/	MAXIMUM
Msg Frequency	2.000E-03	14373	3.561E-05	2.069E-05	0.000	1.640E-04

CELL/	LOWER LIM/	N/	FREQ/	CUM :	
0	-INFINITY	0	0.00	0.00	I
1	0.000	426	0.03	2.96	I**********
2	3.000E-06	222	0.02	4.51	I******
3	6.000E-06	304	0.02	6.62	I*********
4	9.000E-06	453	0.03	9.78	I************
5	1.200E-05	524	0.04	13.42	I***************
6	1.500E-05	648	0.05	17.93	I******************
7	1.800E-05	809	0.06	23.56	I************************
8	2.100E-05	851	0.06	29.48	I************************
9	2.400E-05	957	0.07	36.14	I*****************************
10	2.700E-05	1014	0.07	43.19	I*******************************
11	3.000E-05	1052	0.07	50.51	I********************************
12	3.300E-05	921	0.06	56.92	I**************************
13	3.600E-05	882	0.06	63.06	I**************************
14	3.900E-05	793	0.06	68.57	I**********************
15	4.200E-05	737	0.05	73.70	I**********************
16	4.500E-05	639	0.04	78.15	I******************
17	4.800E-05	527	0.04	81.81	I***************
18	5.100E-05	430	0.03	84.80	I************
19	5.400E-05	337	0.02	87.15	I**********
20	5.700E-05	270	0.02	89.03	I********
21	6.000E-05	212	0.01	90.50	I******
22	6.300E-05	231	0.02	92.11	I*******
23	6.600E-05	166	0.01	93.27	I*****
24	6.900E-05	135	0.01	94.20	I****
25	7.200E-05	120	0.01	95.04	I***
26	7.500E-05	713	0.05	100.00	I********************

S U M M A R Y

TITLE / (RE)SET/	OBS/	AVERAGE/EST.ST.DV/	MINIMUM/	MAXIMUM
Message Delay 2.000E-03	14380	5.134E-05 1.949E-05	1.280E-05	2.564E-04

CELL/LOWER LIM/	N/	FREQ/	CUM :	
0 -INFINITY	0	0.00	0.00	I
1 0.000	0	0.00	0.00	I
2 1.000E-05	141	0.01	0.98	I*
3 2.000E-05	1321	0.09	10.17	I**********
4 3.000E-05	4022	0.28	38.14	I******************************
5 4.000E-05	1062	0.07	45.52	I********
6 5.000E-05	4060	0.28	73.76	I******************************
7 6.000E-05	1815	0.13	86.38	I*************
8 7.000E-05	895	0.06	92.60	I*******
9 8.000E-05	468	0.03	95.86	I***
10 9.000E-05	209	0.01	97.31	I**
11 1.000E-04	179	0.01	98.55	I*
12 1.100E-04	91	0.01	99.19	I*
13 1.200E-04	42	0.00	99.48	I.
14 1.300E-04	35	0.00	99.72	I.
15 1.400E-04	18	0.00	99.85	I.
16 1.500E-04	9	0.00	99.91	I.
17 1.600E-04	4	0.00	99.94	I.
18 1.700E-04	3	0.00	99.96	I.
19 1.800E-04	2	0.00	99.97	I.
20 1.900E-04	1	0.00	99.98	I.
21 2.000E-04	1	0.00	99.99	I.
22 2.100E-04	0	0.00	99.99	I
23 2.200E-04	1	0.00	99.99	I.
24 2.300E-04	0	0.00	99.99	I
25 2.400E-04	0	0.00	99.99	I
26 2.500E-04	1	0.00	100.00	I.

S U M M A R Y

TITLE / (RE)SET/	OBS/	AVERAGE/EST.ST.DV/	MINIMUM/	MAXIMUM
Packet Delay 2.000E-03	50327	3.896E-05 1.889E-05	1.280E-05	2.564E-04

CELL/LOWER LIM/	N/	FREQ/	CUM :	
0 -INFINITY	0	0.00	0.00	I
1 0.000	0	0.00	0.00	I
2 1.000E-05	6186	0.12	12.29	I**************
3 2.000E-05	13168	0.26	38.46	I******************************
4 3.000E-05	12984	0.26	64.26	I******************************
5 4.000E-05	3932	0.08	72.07	I*********
6 5.000E-05	7936	0.16	87.84	I******************
7 6.000E-05	3271	0.06	94.34	I*******
8 7.000E-05	1327	0.03	96.97	I***
9 8.000E-05	699	0.01	98.36	I**
10 9.000E-05	296	0.01	98.95	I*
11 1.000E-04	229	0.00	99.41	I*

12	1.100E-04	123	0.00	99.65	I.
13	1.200E-04	81	0.00	99.81	I.
14	1.300E-04	41	0.00	99.89	I.
15	1.400E-04	23	0.00	99.94	I.
16	1.500E-04	13	0.00	99.96	I.
17	1.600E-04	4	0.00	99.97	I.
18	1.700E-04	7	0.00	99.99	I.
19	1.800E-04	3	0.00	99.99	I.
20	1.900E-04	0	0.00	99.99	I
21	2.000E-04	1	0.00	99.99	I.
22	2.100E-04	0	0.00	99.99	I
23	2.200E-04	1	0.00	100.00	I.
24	2.300E-04	0	0.00	100.00	I
25	2.400E-04	1	0.00	100.00	I.
26	2.500E-04	1	0.00	100.00	I.

S U M M A R Y

TITLE / (RE)SET/	OBS/	AVERAGE/EST.ST.DV/	MINIMUM/	MAXIMUM
Msg Delay D=1 2.000E-03	7509	4.979E-05 1.824E-05	1.280E-05	1.939E-04

CELL/LOWER LIM/	N/	FREQ/	CUM :	
0 -INFINITY	0	0.00	0.00	I
1 0.000	0	0.00	0.00	I
2 1.000E-05	141	0.02	1.88	I**
3 2.000E-05	815	0.11	12.73	I**********
4 3.000E-05	1998	0.27	39.34	I************************
5 4.000E-05	367	0.05	44.23	I*****
6 5.000E-05	2358	0.31	75.63	I******************************
7 6.000E-05	1023	0.14	89.25	I*************
8 7.000E-05	393	0.05	94.49	I*****
9 8.000E-05	191	0.03	97.03	I**
10 9.000E-05	87	0.01	98.19	I*
11 1.000E-04	66	0.01	99.07	I*
12 1.100E-04	32	0.00	99.49	I.
13 1.200E-04	13	0.00	99.67	I.
14 1.300E-04	11	0.00	99.81	I.
15 1.400E-04	8	0.00	99.92	I.
16 1.500E-04	3	0.00	99.96	I.
17 1.600E-04	2	0.00	99.99	I.
18 1.700E-04	0	0.00	99.99	I
19 1.800E-04	0	0.00	99.99	I
20 1.900E-04	1	0.00	100.00	I.

S U M M A R Y

TITLE / (RE)SET/	OBS/	AVERAGE/EST.ST.DV/	MINIMUM/	MAXIMUM
Msg Delay D=2 2.000E-03	5793	5.160E-05 2.049E-05	2.560E-05	2.564E-04

CELL/LOWER LIM/	N/	FREQ/	CUM :	
0 -INFINITY	0	0.00	0.00	I

1	0.000	0	0.00	0.00	I
2	1.000E-05	0	0.00	0.00	I
3	2.000E-05	506	0.09	8.73	I********
4	3.000E-05	1871	0.32	41.03	I******************************
5	4.000E-05	633	0.11	51.96	I**********
6	5.000E-05	1301	0.22	74.42	I*********************
7	6.000E-05	584	0.10	84.50	I*********
8	7.000E-05	382	0.07	91.09	I******
9	8.000E-05	220	0.04	94.89	I****
10	9.000E-05	96	0.02	96.55	I**
11	1.000E-04	91	0.02	98.12	I*
12	1.100E-04	46	0.01	98.91	I*
13	1.200E-04	24	0.00	99.33	I.
14	1.300E-04	18	0.00	99.64	I.
15	1.400E-04	10	0.00	99.81	I.
16	1.500E-04	5	0.00	99.90	I.
17	1.600E-04	1	0.00	99.91	I.
18	1.700E-04	2	0.00	99.95	I.
19	1.800E-04	1	0.00	99.97	I.
20	1.900E-04	0	0.00	99.97	I
21	2.000E-04	0	0.00	99.97	I
22	2.100E-04	0	0.00	99.97	I
23	2.200E-04	1	0.00	99.98	I.
24	2.300E-04	0	0.00	99.98	I
25	2.400E-04	0	0.00	99.98	I
26	2.500E-04	1	0.00	100.00	I.

S U M M A R Y

TITLE	/ (RE)SET/	OBS/	AVERAGE/EST.ST.DV/	MINIMUM/	MAXIMUM
Msg Delay D=3	2.000E-03	979	5.973E-05 1.913E-05	3.840E-05	1.892E-04

CELL/LOWER LIM/		N/	FREQ/	CUM :	
0	-INFINITY	0	0.00	0.00	I
1	0.000	0	0.00	0.00	I
2	1.000E-05	0	0.00	0.00	I
3	2.000E-05	0	0.00	0.00	I
4	3.000E-05	153	0.16	15.63	I************
5	4.000E-05	62	0.06	21.96	I*****
6	5.000E-05	374	0.38	60.16	I******************************
7	6.000E-05	170	0.17	77.53	I**************
8	7.000E-05	101	0.10	87.84	I********
9	8.000E-05	52	0.05	93.16	I****
10	9.000E-05	24	0.02	95.61	I**
11	1.000E-04	19	0.02	97.55	I**
12	1.100E-04	11	0.01	98.67	I*
13	1.200E-04	4	0.00	99.08	I.
14	1.300E-04	6	0.01	99.69	I.
15	1.400E-04	0	0.00	99.69	I
16	1.500E-04	1	0.00	99.80	I.
17	1.600E-04	0	0.00	99.80	I
18	1.700E-04	1	0.00	99.90	I.

19	1.800E-04	1	0.00	100.00	I.

S U M M A R Y

TITLE / (RE)SET/	OBS/	AVERAGE/EST.ST.DV/	MINIMUM/	MAXIMUM
Msg Delay D=4 2.000E-03	90	6.821E-05 2.093E-05	5.120E-05	2.099E-04

CELL/LOWER LIM/	N/	FREQ/	CUM :	
0 -INFINITY	0	0.00	0.00	I
1 0.000	0	0.00	0.00	I
2 1.000E-05	0	0.00	0.00	I
3 2.000E-05	0	0.00	0.00	I
4 3.000E-05	0	0.00	0.00	I
5 4.000E-05	0	0.00	0.00	I
6 5.000E-05	27	0.30	30.00	I************************
7 6.000E-05	36	0.40	70.00	I********************************
8 7.000E-05	16	0.18	87.78	I**************
9 8.000E-05	4	0.04	92.22	I***
10 9.000E-05	2	0.02	94.44	I**
11 1.000E-04	1	0.01	95.56	I*
12 1.100E-04	2	0.02	97.78	I**
13 1.200E-04	1	0.01	98.89	I*
14 1.300E-04	0	0.00	98.89	I
15 1.400E-04	0	0.00	98.89	I
16 1.500E-04	0	0.00	98.89	I
17 1.600E-04	0	0.00	98.89	I
18 1.700E-04	0	0.00	98.89	I
19 1.800E-04	0	0.00	98.89	I
20 1.900E-04	0	0.00	98.89	I
21 2.000E-04	1	0.01	100.00	I*

S U M M A R Y

TITLE / (RE)SET/	OBS/	AVERAGE/EST.ST.DV/	MINIMUM/	MAXIMUM
Msg Delay D=5 2.000E-03	9	9.055E-05 3.161E-05	6.400E-05	1.650E-04

CELL/LOWER LIM/	N/	FREQ/	CUM :	
0 -INFINITY	0	0.00	0.00	I
1 0.000	0	0.00	0.00	I
2 1.000E-05	0	0.00	0.00	I
3 2.000E-05	0	0.00	0.00	I
4 3.000E-05	0	0.00	0.00	I
5 4.000E-05	0	0.00	0.00	I
6 5.000E-05	0	0.00	0.00	I
7 6.000E-05	2	0.22	22.22	I********************
8 7.000E-05	3	0.33	55.56	I******************************
9 8.000E-05	1	0.11	66.67	I**********
10 9.000E-05	0	0.00	66.67	I
11 1.000E-04	2	0.22	88.89	I********************
12 1.100E-04	0	0.00	88.89	I
13 1.200E-04	0	0.00	88.89	I

14	1.300E-04	0	0.00	88.89	I
15	1.400E-04	0	0.00	88.89	I
16	1.500E-04	0	0.00	88.89	I
17	1.600E-04	1	0.11	100.00	I**********